UNIVERSITY OF OSLO
Department of Informatics

Retroactively Parallelizing a Large Python System

Master's thesis

Jonathan Lunde Lillesæter

May 5, 2011

Contents

List of Figures

Abstract

Computers today become more powerful through increased numbers of processors rather than clock speed increases as in the past. Exploiting this paralellism requires different software design strategies than do sequential programs.

The immense increase in the generation of genomic scale data poses an unmet analytical challenge, due to a lack of established methodology with the required flexibility and power. The Hyperbrowser is a framework for comparative analysis of sequence-level genomic data and aims to solve this problem. It is currently a single-threaded system, and in order to both be able to scale better and to reduce the analysis time, parallelization is desireable.

A flexible framework for distributing compute intensive, independent tasks over many computers is presented. The framework allows for many concurrent users, and exploits both local and remote computing resources. This framework is used to achieve significant speedups for analyses in the Hyperbrowser, both due to parallelizing the workload and due to exploiting the Titan compute cluster.

Performance tests show that the framework is fairly efficient for both large and small jobs and scales well. A number of possible future improvements are suggested.

Acknowledgements

First and foremost, I would like to thank my supervisor Geir Kjetil Sandve for all of his help and guidance. I also want to thank my co-supervisors Eivind Hovig and Torbjørn Rognæs for valuable feedback.

Additionally, my thanks go to family and friends, especially the rest of the team at the tenth floor of Ole Johan Dahls hus.

Jonathan Lunde Lillesæter
University of Oslo
May 2011

Chapter 1

Introduction

1.1 Motivation

The Genomic HyperBrowser is a system for statistical analysis of genomic data. The rate at which genomic data is being generated is increasing daily. However, performing analyses of these large datasets with the HyperBrowser, especially Monte Carlo simulations with a high number of resamplings, can take a very long time. Several days or even weeks is not uncommon for larger analyses. Speeding this up is naturally desirable — less time spent waiting on results is more time for actual investigation of the results. The HyperBrowser analyses are currently single-threaded. As will be shown later, the Hyperbrowser handles problems that are for the most part embarassingly parallel, therefore making it an ideal candidate for parallelization in order to achieve the desired speedup.

The load on the Hyperbrowser is increasing as it becomes more frequently used. A non-distributed setup is likely not viable in the long run. Being able to offload the computationally intensive tasks to the computing cluster Titan available to researchers at the University of Oslo will, when combined with the parallelization, both increase the possible speedup as well as enable more users to concurrently use the Hyperbrowser.

1.2 Research questions

In this thesis we consider the following research questions.

1. Can a large-scale Python program designed without parallelism in mind effectively be parallelized in order to exploit today's multi-processor architectures? If so, can it be done without massive changes to the existing code base?

2. Is offloading work to a high-performance compute cluster a viable way of speeding up execution times and increasing the available computing

power? If so, can it be done in a way transparent to the user?

3. Compute clusters are designed for maximum throughput, rather than quick response. Can an interactive system where small jobs run alongside larger jobs still yield short execution times for the small jobs when using a computer cluster?

1.3 Chapter overview

Chapter 2 presents the background necessary to fully understand the rest of the thesis.

Chapter 3 describes how a parallel design methodology is applied to the HyperBrowser in order to yield a parallel design.

Chapter 4 shows hows this design has been implemented. Other implementation specific issues are also discussed.

Chapter 5 presents results comparing the performance of the serial program and the parallel version.

Chapter 6 discusses these results. It also contains a discussion on the various design choices taken and implementation details.

Chapter 7 presents a number of suggested future improvements to the system.

Chapter 8 concludes this thesis by giving a summary of the work presented.

Chapter 2

Background

2.1 The Genomic HyperBrowser

The Genomic HyperBrowser is a system for statistical analysis of genomic data. Genomic data is being generated at an unprecedented scale as high-throughput sequencing techniques continue to improve. The HyperBrowser offers statistical analysis of this sequence-level genomic information[24], and provides this functionality through an easy-to-use web interface[1].

It is mostly implemented in Python, but uses a two-level architecture in order to improve performance, as Python is relatively slow compared to languages like C++. At the highest level, Python objects and logic are used to provide flexibility and fast development. At the base-pair level, data are handled as low-level vectors (tracks), allowing efficient indexing and the use of vector operations for speed.

The HyperBrowser is a fairly large system — it consists of over 40000 lines of Python code, not including the third party modules it uses. To aid in the understanding of such a large system, the following section will explain the overall architecture and show the main program flow.

2.1.1 Galaxy

Architecture

The HyperBrowser is based on the Galaxy framework [14][6], a project that aims to provide experimental biologists with simple interfaces to powerful computational tools. It allows biologists without any informatics or programming knowledge to perform complex large-scale analysis of genomic data within their own web browser. The Galaxy framework handles encapsulation of the complicated, high-end computational tools that are needed to perform large-scale analyses, and presents the user with an intuitive web interface. Rather than getting bogged down with technical details like storage management and hard to use command line tools, the biologists can focus

on what is important: their actual research questions.

In addition to allowing experimental biologists without extensive knowledge of computer science to perform complex genomic analysis, it also makes reproduction of results far easier. Especially when many complex computational tools are used in a workflow (often with raw data that may not be readily available), verifying the results can be difficult. Galaxy allows workflows (a set of steps that describes the computational analysis being carried out) to be saved, for which a link can then be provided to other researchers so that they can review the methodology used.

For developers, Galaxy has an easy-to-use system for adding functionality, through the *tool* model Galaxy uses. A tool can be any piece of software (written in any kind of language), as long as it has a command line interface. For Galaxy to be able to use this tool, all that is required is writing a configuration (XML) file that describes how the tool is to be run, as well as a specification for input and output. Galaxy uses this information to work with the tool in an abstract way, automatically generating a web interface. The HyperBrowser analysis tool is implemented as such a tool.

The HyperBrowser uses Galaxy for its web front-end, building on Galaxy's core philosophy to provide researchers without a computer science background an easy to use platform for statistical analysis of genomic data. Instead of getting bogged down with configuration issues and spending time on deciding what tools to use in what order, the researchers can spend more time on getting the question asked (the hypothesis) right in an intuitive and simple manner.

Figure 2.1 shows the HyperBrowser web interface with the analysis tool ready to perform an analysis of the relation between H3K27me3 histone modifications and SINE repeats in the mouse genome. The user is not explicitly asked about what tools he or she wants to use, how they should be used, where the data files are located on the disk, or other unnecessary technical questions. Instead he or she constructs a statistical hypothesis by selecting two annotated genomic tracks and defining a question regarding how these two tracks relate. For each kind of question or analysis, only the relevant options are shown depending on the tool that needs to be run to perform this analysis. All of this is done dynamically based on the tool configuration files.

Figure 2.2 shows the results after running the analysis in Figure 2.1. The results clearly show exactly the kind of analysis that has been performed, both with a simple answer as well as a much more detailed answer. The detailed answer includes a detailed description of what the hypothesis and null hypothesis were, the rules used for preservation or randomization of tracks, and so on. Results are available in raw data form, HTML form and for most analyses various plots are created as well (scatter plots, graphs,

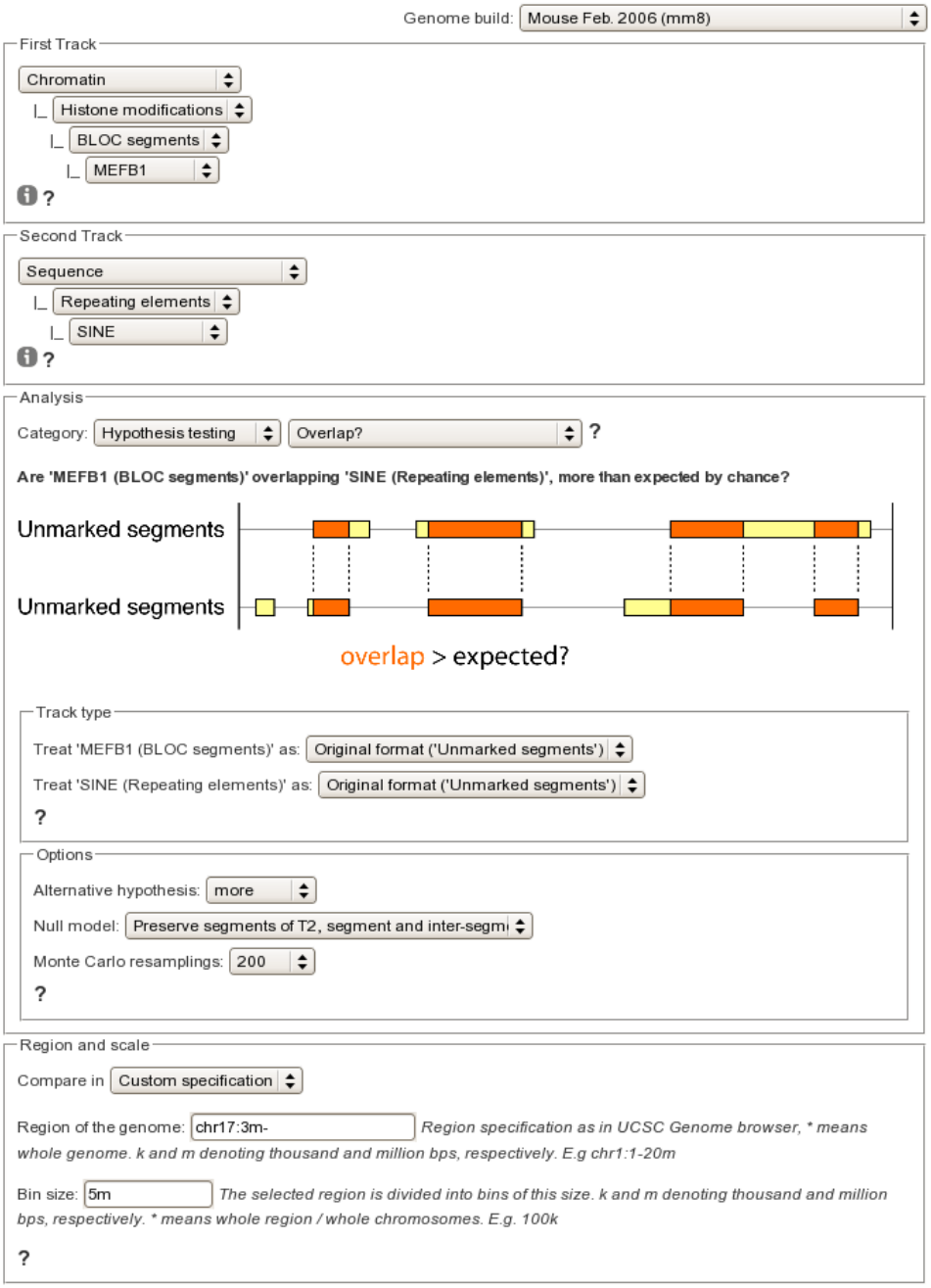


Figure 2.1: The HyperBrowser user interface. Analysis of the relation between H3K27me3 histone modifications and SINE repeats in the mouse genome.

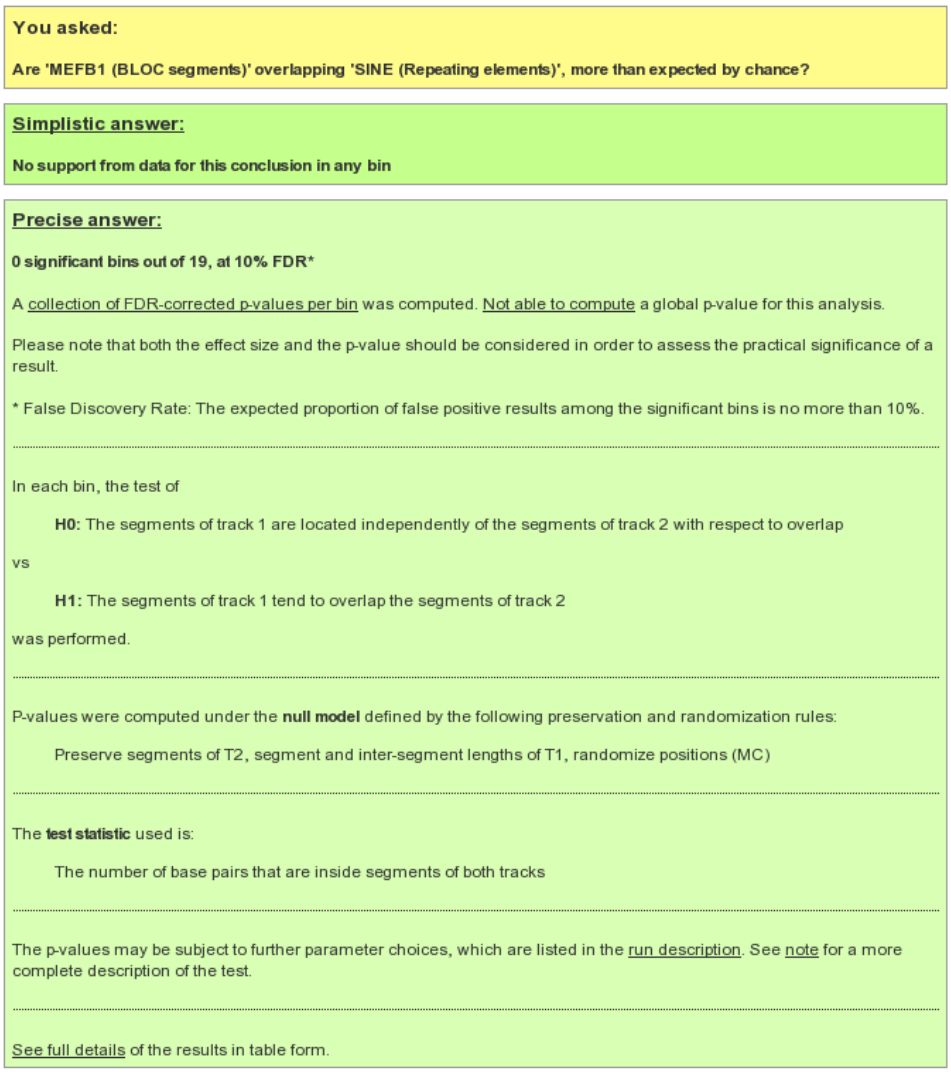


Figure 2.2: An example of results from an analysis in the HyperBrowser. The results are from the analysis described in Figure 2.1.

heatmaps and so on).

Running Galaxy

Galaxy typically runs on a server computer like any other web service. However, with data analysis requirements often varying widely over time, it can also run on cloud services[5] such as the Amazon Elastic Compute Cloud service [1] or Eucalyptus [2]. This can save researchers from having expensive hardware standing unused most of the time.

Galaxy jobs

A *job* is a Galaxy tool that is being run. Each job runs in its own independent process which is spawned by the main Galaxy process. This allows multiple jobs to run concurrently on the same Galaxy instance.

There is support for offloading jobs onto compute clusters via DRMAA, an "an API specification for the submission and control of jobs to [...] Distributed Resource Management (DRM) systems" [2]. An implementation of this exists for Simple Linux Utility for Resource Management (SLURM) (see section 2.3.1) [3].

2.1.2 HyperBrowser design

The HyperBrowser operates on *tracks*, which are split into *bins*. Tracks, short for "genomic annotation track", are collections of objects for a specific genomic feature, such as genes, with base-pair specific locations. A bin can be thought of as slices of a track. Tracks are divided into bins during computation. The length of a bin is determined by the user in the analysis specification, and is is expressed as how many base pairs (abbreviated bp) the bin covers. See Figure 2.1.2

For each bin in a track, a statistic is constructed and computed, before the results from each bin are combined to yield a global result.

2.1.3 HyperBrowser job execution

The HyperBrowser performs its statistical analyses in two phases: first a local, then a global analysis.

> A global analysis investigates if a certain relation between two tracks is found in a domain as a whole. A local analysis is based

[1] "Amazon Elastic Compute Cloud (Amazon EC2) is a web service that provides resizable compute capacity in the cloud. It is designed to make web-scale computing easier for developers." `http://aws.amazon.com/ec2/`

[2] "An open source software infrastructure for implementing a private cloud using an organization's own information technology." `http://www.eucalyptus.com/`

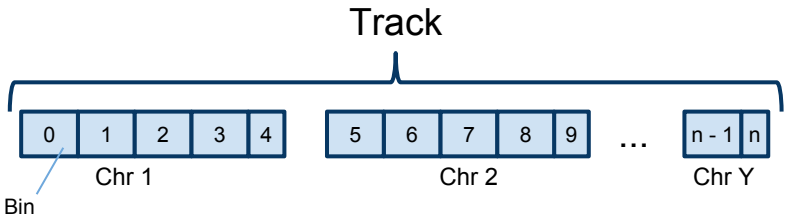


Figure 2.3: Tracks and bins in the HyperBrowser

> on partitioning the domain into smaller units, called bins, and performing the analysis in each unit separately. Local analysis can be used to investigate if and where two tracks display significant concordant or discordant behavior [...] Local investigations may also be used to examine global results in more detail. The length of each bin defines the scale of the analysis.
>
> (From [24])

2.1.4 Job submission

A job is usually initiated from the Galaxy web interface (figure 2.1), where users can select which genome dataset to work on, which tracks to use in the analysis, and which hypothesis they wish to test. A job may also be initiated from a batch interface, where jobs can be described in a textual way, similar to a command line tool. This is mostly used for testing, for example running a new statistic on many different tracks.

2.1.5 Code flow

Jobs can be said to start in `GalaxyInterface`, which as the name suggests is the interface between the HyperBrowser and Galaxy. The various necessary job arguments, like tracks, statistical test, binning specification and so on, are parsed and used to prepare the run. Track names and the analysis definitions often require some cleanup before it can be passed to other parts of the program. The cleaned up specifications are used to create a `StatJob` object, which is more or less a container for the actual objects used in the computation. `Track` objects and the top level `Statistic` object is created here. What a "top level" statistic is will be explained shortly.

Once the preparations are done, `run` is called on the `StatJob` object. This call will not return until the results are ready. Once the call returns, `GalaxyInterface._handleRunResults` is called. This method parses the results and creates a web page with the results, presented in graph form, tables etc. depending on what kind of analysis was performed. This web page is then presented to the user.

Computation flow

The following sections include UML 2.0 diagrams to ease the understanding of the HyperBrowser. See [12] if not familiar with UML or the relatively new 2.0 standard.

`StatJob.run` starts the actual computation. Progress tracking is initialized before the computational phases begin. First `_doLocalAnalysis` is called, then `_doGlobalAnalysis`, for the different phases described in 2.1.3.

`_doLocalAnalysis` iterates over the user-defined bins and for each of them calls `_getSingleResult`, which creates a `Statistic` object that corresponds to the analysis being carried out. `getResult` is then called on this `Statistic` object, which computes the result for the bin in question. Once the result has been computed for every bin, `getResult` returns.

`_doGlobalAnalysis` is carried out after the local analysis has taken place. As the results it requires has already been computed by the local analysis it fetches them from the memoization lookup table (see Section 2.1.7).

2.1.6 Statistic objects

A *statistic* represents a statistical analysis for a specific bin and track (or tracks, if the analysis being performed compares two tracks).

Terminology

Throughout the thesis the term statistic is often used. Unless otherwise noted, this refers to a module in the HyperBrowser that defines a statistical test which operates on tracks. The terminology can get somewhat complicated; in each of these statistic submodules (from now on just called *statistics*), at least two classes are defined: a *unsplittable* implementation class and a *factory* class. This unsplittable implementation class is always a subclass of the `Statistic` superclass, and the factory class is always a subclass of `MagicStatFactory` (see Section 2.1.7). Some statistics also have a *splittable* class.

Naming scheme All statistics have a "base name" that is the same as the actual module name, with the naming scheme of `"«description»" + "Stat"`. Each statistic has two classes, with the naming scheme `"«base name»" + {"Splittable", "Unsplittable"}`. Let us use the simple statistic `MeanStat` as an example. The description of what it does is "mean", as it simply calculates the mean of a set of data points in a track. Its base name is therefore `"Mean" + "Stat" = "MeanStat"`, which is both the name of the module, and of the factory class used to produce statistics of this type. It has both splittable and unsplittable implementations, these are respectively named `"MeanStatSplittable"` and `"CountStatUnsplittable"`.

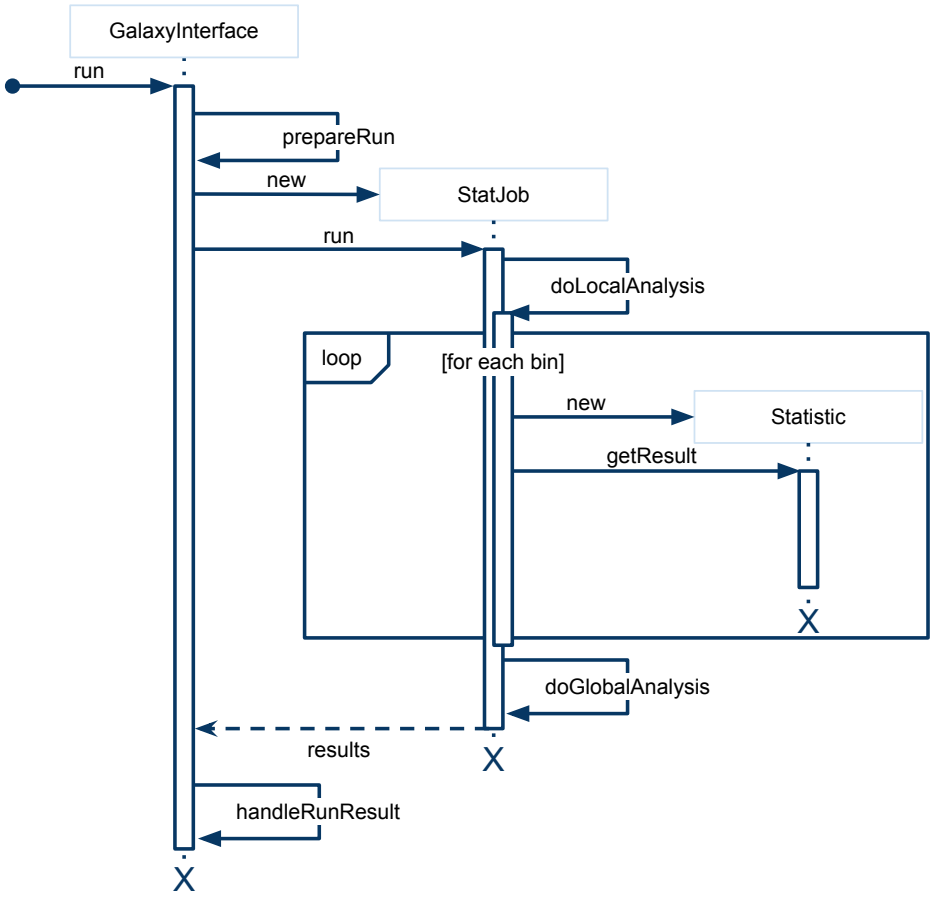


Figure 2.4: A slightly simplified UML 2.0 sequence diagram of the main flow in the execution of a HyperBrowser job.

Splittable and Unsplittable statistic classes All statistics have at least an unsplittable class that is the actual implementation class. This is the class that contains the code for the computation of the statistic.

Some statistics also have a Splittable version: for certain statistics, it is possible to split the problem into smaller subproblems. For example, counting the number of points in a bin is a problem that can easily be split up into smaller subproblems: simply count the number of points in sub-bins, and then add the results together for a total count over the whole bin. For other problems, this cannot be done, for example finding the mean (each bin can have a different number of points, which would make combining the means of several bins impossible). The main reason for this is memory. Take for example the statistic `CategoryPointCountInSegsMatrixStat`. Behind the somewhat cryptic name lies a statistic that creates a NumPy matrix during computation, with base pairs along one axis and categories along the other. Exactly what the matrix is used for is not important, but the size of the matrix is: the statistic is commonly used in analyses where entire chromosomes are used as bins, with tracks that have well over 1000 categories. As for example chromosome 1 is 250 million base pairs, if using a matrix of the boolean data type[3], the matrix would consume 232 gigabytes of memory when analyzing chromosome 1 and tracks with 1000 categories. Splitting this into smaller sub-bins makes the memory use much more manageable; with splitting the bin into 100Kbp (the default value) sub-bins, the matrix will consume a more pleasant 95 megabytes.

Worth noting is that the splittable classes do still use the unsplittable implementation classes; a splittable class will create children that are unsplittable versions of itself. The splittable class defines *how* the statistic can be split up into smaller problems, not how the statistic is actually *computed*; the unsplittable version handles that.

Required methods for statistics

All statistic implementation classes must implement two methods: one that computes the result of the statistic (`_compute`) and one that describes its relation to other statistics (`_createChildren`). To understand what these do it is important to first understand the general computation strategy the HyperBrowser uses.

Most statistics are based on a directed acyclic graph of children statistics. This reflects the fact that that most statistical measurements rely on a number of more "basic" measurements. For example, to find the standard deviation one needs the variance, and to find the variance one needs the mean, and so on. In order to express the same relationship in terms of source code, the `_compute` and `_createChildren` methods are used.

[3] In NumPy, the smallest possible data type requires one byte of memory, even boolean values that in an ideal world would only require one bit.

As said, for a statistic to be valid it must implement two methods: `_createChildren` and `_compute`. The `_compute` method defines what the class actually *computes*. The result of this computation is stored in the statistic instance attribute `_result`. The `_createChildren` method defines what *child* statistics it requires in order to be able to compute its results.

As an example, let us use the simple statistic `MeanStat`, shown in Listing 2.1.

Listing 2.1: MeanStat: An example of a simple statistic

```
Line 1  ...import MagicStatFactory, Statistic, Countstat, SumStat...
-
-   class MeanStat(MagicStatFactory):
-       pass
5
-   class MeanStatUnsplittable(Statistic):
-       def _compute(self):
-           return 1.0 * self._children[0].getResult() \
-                      / self._children[1].getResult()
10
-       def _createChildren(self):
-           self._addChild(SumStat(self._bin, self._track))
-           self._addChild(CountStat(self._bin, self._track))
```

To calculate the mean of a given sequence of numbers, you need to know the length of the sequence as well as the total sum of the sequence. Therefore this statistic declares that `CountStat` and `SumStat` are its children to express this dependence.

Computing the results of a statistic

Computing the result of a statistic is done by by calling the `getResult` method. This method checks to see if the result has already been computed (to account for the fact that the result could be stored in the memoization lookup table, see below). If the results are not already stored within the instance, it begins the computation by calling `compute` on itself. The `compute` method creates the required child statistics by calling `createChildren`. The `createChildren` method recursively instantiates children statistics, in order to construct a statistics tree like shown in Figure 2.5. The `compute` method is then called recursively on each of the children, before computing and returning its own result.

This is more easily demonstrated with a simplified code example:

Listing 2.2: A simplified version of `_compute`.

```
Line 1  def compute(self):
-           if self.hasResult():
-                   return self.result
-
```

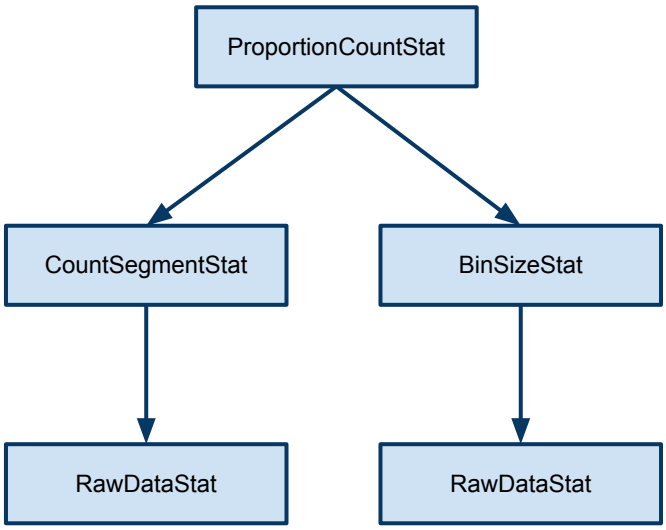


Figure 2.5: A graph consisting of the necessary statistics to compute the proportional coverage of a track, for example finding out how much of the genome is covered by genes. Note the statistic `RawDataStat`, which can be found at the bottom level of all statistics graphs. This is not really a "true" statistic; it handles reading in data from disk. It is implemented as a statistic to memoize data read from disk. All statistics that require direct file access uses `RawDataStat`.

```
5           self.createChildren()
-           for child in self.children:
-                   child.compute()
-
-           #now the children contain the necessary
10          #result to compute this statistic, or this
-           #statistic is one without children and
-           #does not rely on results from other
-           #statistics
-           return self.computeResult()
```

2.1.7 Memoization

When statistic objects are created, such as by the `createChildren` method, the objects returned may not be new objects. The HyperBrowser uses a *memoization* scheme to prevent the same result being computed several times in the same run. Memoization is a powerful and fundamental optimization technique for result re-use and is primarily used to speed up computer programs. In short it is having function calls store their results in a lookup table

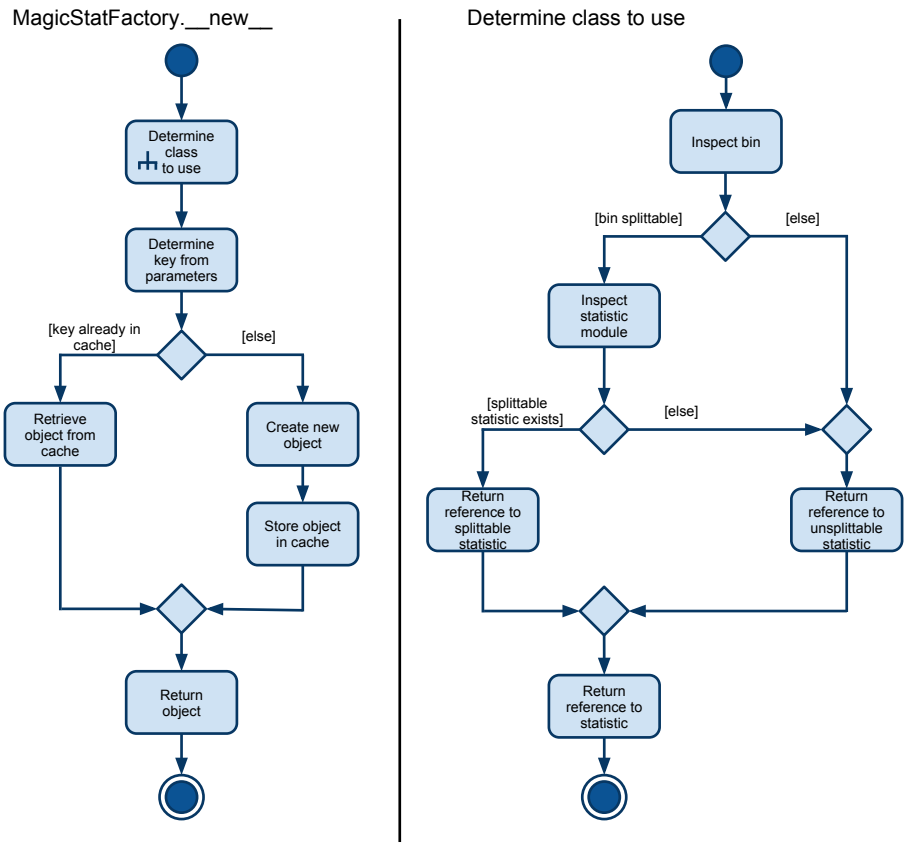


Figure 2.6: Activity diagram of creating new statistics. Statistics inherit their constructors from `MagicStatFactory`. UML 2.0.

and checking this lookup table when called to avoid repeating the calculation of results for previously processed inputs.

In the HyperBrowser, memoization is employed in a manner transparent to the rest of the system for all statistics. It is implemented in a fairly convoluted manner which utilizes Python techniques perhaps not commonly seen.

As explained in Section 2.1.6, statistics have a "base class". This is the class name that is used by other parts of the system when a statistic of that type is required, for example in the `_addChildren` all statistic implementation classes define. This "base class" is a subclass of `MagicStatFactory`. As the name implies, this is an implementation of the factory method design pattern (a method used to create other objects, an abstraction of the constructor)[13]. By doing this they inherit the `__new__` method from `MagicStatFactory`. The `__new__` method in `MagicStatFactory` examines the arguments passed

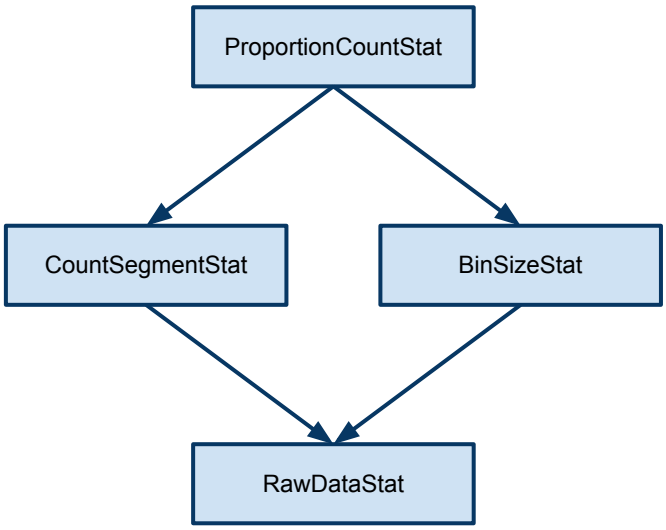


Figure 2.7: As Figure 2.5, but demonstrating how statistics reuse already computed substatistics.

to it and uses it to determine a unique key. This key is a tuple that contains the Statistic's name in string form, the bin, track names as well as any optional keyword arguments. This key is then used to perform a lookup in a weak reference dictionary (see Section 2.4.1) contained in the `MagicStatFactory` module. In practice, what this means is that if a new statistic object is about to be created with the same arguments as an object made earlier, a reference to this object is returned instead. What this accomplishes is that results from already completed statistics do not have to be computed several times during one run.

If the object is *not* found in the dictionary, a new instance is made. This is also done in a somewhat convoluted manner. Let us use CountStat as an example. First the bin to be used in the statistic is inspected. If it can be split into smaller bins the statistic to be used is inspected for "splittability". The base class name (the name of the class that subclasses `MagicStatFactory`, so `"CountStat"` in this example) is concatenated with `"Splittable"` and refsection is then used to see if this class exists. If it does not, the statistic is assumed to not support splitting, and an instance of `"«base name»"` + `"Unsplittable"` is created and returned. If it *is* splittable (continuing with our example, `CountStat` does indeed have a `CountStatSplittable` class, and is thus splittable), an instance of `"«base name»"` + `"Splittable"` is created and returned.

This process is complex, and can more easily be understood through a figure; see Figure 2.6 for an overview.

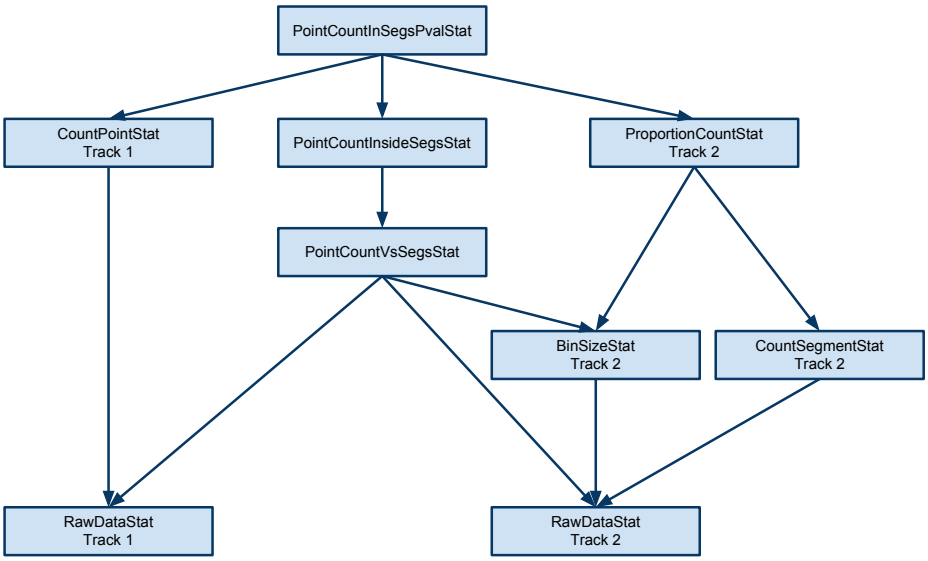


Figure 2.8: An example of a more advanced statistic.

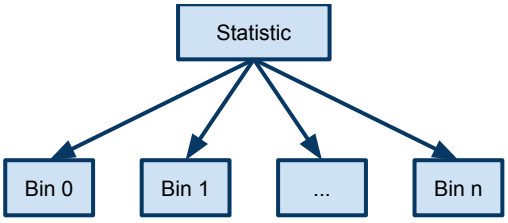
The reason for `MagicStatFactory` using weak references is memory usage. If all Statistic objects that are created were to be stored in a normal dictionary in the MagicStatFactory, manual memory handling would be required. That is, if a statistic object is created and is only required for a short amount of time, it would have to be manually removed from the dictionary (otherwise the reference in the dictionary would be enough to keep it from being garbage collected).

2.1.8 Monte Carlo analyses

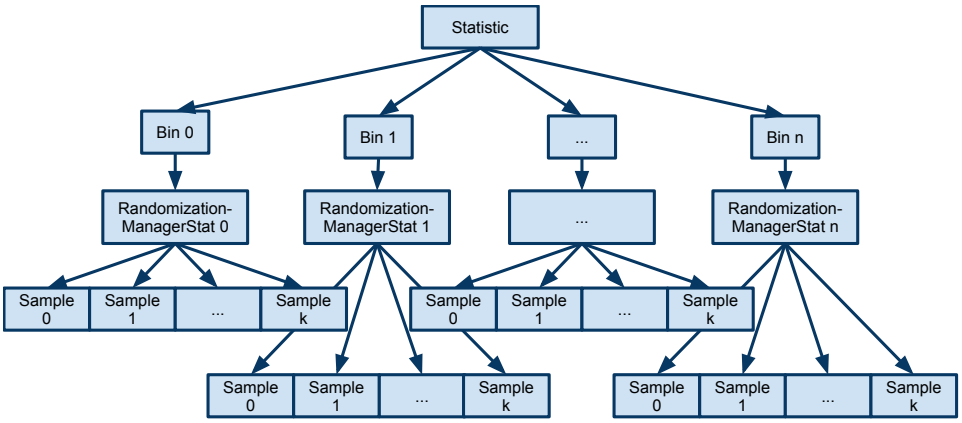
The HyperBrowser often uses Monte Carlo resampling for hypothesis testing in analyses. This is implemented in a somewhat peculiar way that it deserves special mention — it has led to some special design considerations when parallelizing, as will be shown later.

Monte Carlo methods are a type of computational algorithms that rely on repeated random sampling. They are often used in simulating mathematical systems.

RandomizationManagerStat Monte Carlo is implemented as a special statistic called `RandomizationManagerStat`. This statistic is constructed so that the children statistics are not statically defined in `createChildren` like for normal statistics. Instead, it is defined during program execution by sending the name of the statistic as an initialization parameter to



(a) A normal statistic. n is the number of bins.



(b) A Monte Carlo statistic. n is the number of bins, k the number of samples per bin.

Figure 2.9: The difference between a Monte Carlo and a non-Monte Carlo statistic.

`RandomizationManagerStat`.

During computation, for each resampling new instances of this child statistic (which has child statistics of its own) are created. New randomized tracks are then created and passed to these instances for computation. So, for a Monte Carlo analysis with n resamplings, n child statistics are created.

Note that this is all on a per-bin basis, so that one `RandomizationManagerStat` is made per bin, and in effect becomes the "top level" statistic, as explained in Section 2.1.6.

Figure 2.1.8 graphically demonstrates the difference between a normal statistic and one using Monte Carlo.

2.2 Parallel programs

2.2.1 On parallel computing

Parallel computing is computing where many calculations are carried out at the same time ("in parallel"), as opposed to performing the calculations one after another (sequential execution). The key principle is that many large problems can be subdivided into smaller problems, which can then be solved concurrently in order to reduce the execution time. Parallelism can be found in both hardware and software.

Parallelism in software

In software one separates between parallelism in the operating system and in user programs. To the user it looks like several programs execute on a computer simultanously, even if it only has one processor. Even though the system has more than one processor available, as most modern machines today do have, there are almost always many more processes running at the same time than there are processors available. What provides the user with this illusion of concurrent execution is the fact that the operating system treats processor time like any other resource that can be shared among programs, like memory, disk access and so on. In order to do this, processor time is divided into *time slices*. Processes that need processor time are put in a queue, and are given a time slice when possible, subject to the operating system's *scheduling algorithm*. If the process is still running by the time its allotted time slice is over, it is stopped and swapped out by the operating system, without the process knowing about it. To the user, this makes it look like many programs are executing at the same time. However, it is not *true* concurrency; everything is still sequential. From this point on, whenever the terms *parallel*, *concurrent* or the like are used, *true* concurrency is implied, unless otherwise noted.

Today most truly parallel programs use *threads* for their concurrency. A thread can be defined as an independent stream of sequential instructions. Threads run within a process, and do not require as many resources to create as do processes. Switching between threads is also much faster than switching between processes. Threads within a process share memory address space, so additional care must be taken to ensure that threads do not simultanously try to access the same memory or rely on some shared state. To avoid this, *mutexes*, short for "mututal exclusion" and also known as *locks*, are used to ensure that only a single thread is accessing a shared variable at a time. However, for Python programs using threads for parallelism is usually not a good choice, see Section 2.4.1.

Parallelism in hardware

In hardware there are several different ways of handling parallelism. Michael J. Flynn identified four different types of hardware architectures in 1966, which is known as *Flynn's taxonomy*[10]. It is a classification of computer architectures, and is arguably the most commonly used classification of parallel computers. The classification is based upon the number of concurrent instruction and data streams available in the architecture. Each of these streams can have only one of two possible states: single or multiple.

	Single Data	Multiple Data
Single Instruction	SISD	SIMD
Multiple Instruction	MISD	MIMD

Single Instruction, Single Data (SISD) This type of computer does not have support for parallel execution in any way. A traditional, previous generation single-core computer is a SISD computer. The CPU handles one instruction stream per clock cycle. This used to be the most common computer until fairly recently; however today almost all CPUs are multi-core.

Single Instruction, Multiple Data (SIMD) A type of parallel computer in which all processors execute the same instruction per clock cycle, but each processor can act on a different data element. Most computers today employ SIMD instructions due to the Graphics Processor Units (GPUs) coprocessor that many computers have today. As GPUs are specialized for image processing, SIMD instructions are ideal (for example doing one operation for every pixel in an image).

Multiple Instruction, Single Data (MISD) A parallel computer in which, all processors operate on the same single data stream. Each data element is operated upon independently. This architecture is very rare, as it is hard to actually think of a use case for this type of computer that is not already just as easy to do with either SIMD or MIMD.

Multiple Instruction, Multiple Data (MIMD) This type of computer is the most common today. Each processor can work on different instruction streams and different data streams at the same time. Multi-core computers, most current supercomputers and parallel computer clusters are examples of this architecture.

In software, the operating system controls access to the resources available on the machine and manages the running processes. A process is an instance of a computer program that is being executed. Each process has its own address space; a process can only see and modify its own data in memory, not that of other processes.

Processes allow the operating system to share processor time and other resources between programs executing at the same time.

Parallel programs are inherently more difficult to design and implement than single-threaded, serial programs. More care must especially be taken during the design phase, or concurrency problems can easily arise and cause errors that are very hard to pin down, both during the implementation phase and when the program is in production.

In the following section design strategies for parallel programs will be presented.

2.2.2 Designing parallel programs

Foster [11, p. 28] describes a methodology for designing parallel programs, which identifies a four-stage parallel algorithm design process. This approach is intended to identify machine-independent issues such as concurrency early, leaving machine-dependent design aspects until as late in the design process as possible. The methodology structures the design process into four stages: Partitioning, Communication, Agglomeration, Mapping (PCAM). In the first two stages the focus is on concurrency and scalability while in the latter two stages locality is the key issue. The stages are illustrated in Figure 2.10. While the design process is presented here as a sequential activity, in actuality it is a highly parallel process in which many concerns are simultanously being considered. Backtracking, while not ideal, is not avoided at all costs.

1. *Partitioning.* Both the computation that is to be performed as well as the data that is operated on by this computation are decomposed into tasks. Attention is focused on finding opportunities for parallel execution.

2. *Communication.* The necessary communication required for the asks to coordinate is identified, and the communication algorithms and structures required for this is defined.

3. *Agglomeration.* With the task and communication definitions identified in the first two steps in place, the design is evaluated with both performance and implementation costs in mind. Tasks are combined into larger tasks if necessary in order to improve performance or reduce development cost.

4. *Mapping.* Tasks are assigned to processors. Both minimizing communication and maximizing processor utilization is the goal here. Mapping can be static or dynamic (i.e. a load-balancing algorithm that maps tasks to processors at runtime).

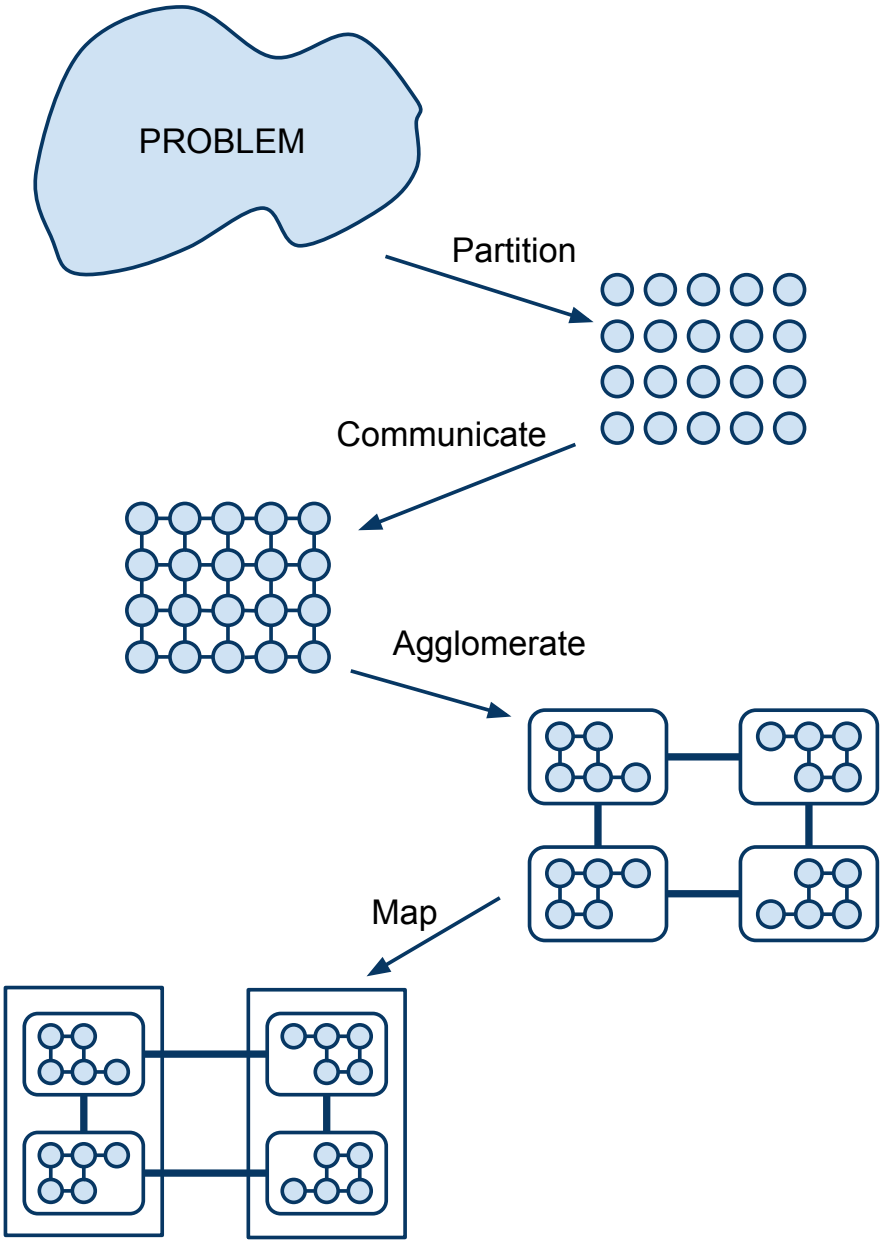


Figure 2.10: A design methodology for parallel programs. Adapted from [11, p. 29].

In the following section I will give a more detailed explanation of these steps.

Partitioning

In this stage, the goal is to identify a *fine-grained*[4] decomposition of the problem in question. A large amount of small tasks is preferrable to fewer, larger tasks at this point, as it is easy to later on in the later design stages to agglomerate many small tasks into larger ones if deemed advantageous. Making the tasks as small as possible in this stage allows for greater flexibility. Two partitioning approaches exist: domain decomposition and functional decomposition. Domain decomposition focuses on the data associated with a problem first and determines a fitting partition before figuring out how to link the computation up data. Functional decomposition instead focuses on the computation first before dealing with the data. Often both of these techniques are applied to the problem in question, as different problem components are probably more suited to one technique than the other.

Domain decomposition In this approach to decomposing the parallel problem, the focus is firstly on the data being operated upon. We seek to divide the data into small pieces of roughly the same size. After partitioning the data, the computation is partiotioned, usually by associating operations with the data on which it operates. This yields a number of tasks, each consisting of some data and a set of operations that are to be carried out on that data.

The rule of thumb is to focus on the most frequently accessed data structure, or the largest (in size). If the computation has several phases (for example and the data requirements are different each phase, we treat the phases separately and later determine how the various phases interface with each other.

Functional decomposition As the name implies, in this approach the focus is on the computation itself rather than the data on which the computation requires. Only fter dividing the computation into separate tasks do we examine the data requirements of these tasks. If the data requirements of these tasks significantly overlap, domain composition should probably be used instead of functional decomposition, in order to avoid unnecessarily large communication costs.

While focusing on the data used in a computation seems like the most obvious way of decomposing a problem in most cases, thinking about the decomposition in terms of the computation itself without caring about the

[4]Decomposition of a problem into a large number of small tasks is called fine-grained, as opposed to *coarse-grained* where the problem is decomposed into a small number of large tasks. This is known as the *granularity* of the problem.

data is valuable in itself as a different way of looking at the problem. It should therefore be given some through even though domain decomposition clearly seems like the obvious solution.

Communication

Tasks generarated by a partition cannot, in most cases, execute completely independently of each other. If they can, we call the problem *embarassingly parallel*.

Typically, tasks require data associated with another task. Data must be communicated between tasks to allow computation to proceed; this is the focus of the *communication* design phase.

Communication requirements vary widely. Foster ([11]) identifies four communication patterns: local/global, structured/unstructured, static/dynamic and synchronous/asynchronous.

In *local* communication, each task communicates with its "neighbours". For *global* communication, each task has to communicate with many other tasks.

In *structured* communication, the communication forms a structure like a grid, tree or ring. *Unstructured* communication, on the other hand, may be arbitrary graphs.

In *static* communication, the identity of the communication partners does not change over time. For *dynamic* communication, however, the communication structures may be determined by data computed at runtime and may vary widely.

In *synchronous* communication, producers and consumers communicate in a coordinated fashion. In contrast, with *asynchronous* communication the consumers may request data from producers at arbitrary points in time as required.

Agglomeration

Agglomeration is the process of combining smaller tasks into larger ones, in order to improve performance. If the previous two stages of the design process partitioned the problem into a number of tasks that greatly exceeds the number of processors available and the computer is not specifically designed for handling a huge number of small tasks (some architectures, like GPUs, handle this fine and indeed benefit from running millions or even billions of tasks), the design may be highly inefficent. Commonly, this is because tasks have to be communicated to the processor or thread that is to compute said

task. Most communication has costs that are not only proportional with the amount of data transferred, but also incurs a fixed cost for every communication operation (like the latency inherent in setting up a TCP consection). If the tasks are too small, this fixed cost can easily make the design inefficient.

Mapping

In the *mapping* stage of the parallel algorithm design process, we specify where each task is to be executed. The goal is to minimize the total execution time. Here one must often make tradeoffs, as the two main strategies often conflict:

1. Tasks that communicate frequently should be placed on the *same* processor, to increase locality.

2. Tasks that can execute concurrently should be placed on *different* processors, to enhance concurrency.

This is known as the *mapping problem*, and it is known to be *NP-complete* [7]. As such no polynomial time solutions to the problem in the general case exist. For tasks of equal size and tasks with easily identified communication patterns, the mapping is straightforward (we can also perform agglomeration here, to combine tasks that map to the same processor). However, if the tasks have communication patterns that are hard to predict or the amount of work varies per task, easily designing an efficient mapping and agglomeration scheme is hard. For these types of problems, load balancing algorithms can be used to identify agglomeration and mapping strategies during runtime. The hardest problems are those in which the amount of communication or tasks changes during the execution of the program. For these kinds of problems, dynamic load balancing algorithms can be used, which runs periodically during the execution.

Dynamic mapping There exists many load balancing algorithms for various problems, both global and local. Global algorithms require global knowledge of the computation being performed, which often adds a lot of overhead. Local algorithms rely only on information local to the task in question, which reduces overhead compared to global algorithms, but are usually worse at finding an optimal agglomeration and mapping. However, the reduced overhead may reduce the execution time, even though the mapping is worse in itself.

If the tasks rarely communicate, other than at the start and end of the execution, a task-scheduling algorithm is often used that simply map tasks to processors as they become idle. In a task-scheduling algorithm, a task pool is maintained. Tasks are placed into this pool and taken from it by *workers*. There are three common approaches with this model.

Manager/Worker This is the basic dynamic mapping scheme, in which all the workers connect to a centralized manager. The manager repeatedly sends tasks to the workers and collects the results. This strategy is probably the best for a relatively small number of processors. The basic strategy can be improved through prefetching of tasks so that communication and computation overlaps.

Hierarchical Manager/Worker A variant of manager/worker that has a semi-distributed layout; workers are split into groups, each with their own manager. These group managers communicate with the central manager (and possibly amongst themselves as well), while workers request tasks from the group managers. This spreads the load amongst several managers and can as such handle a larger amount of processors than if all workers request tasks from the same manager.

Decentralized In this scheme, everything is decentralized. Each processor maintains its own task pool, and communicates with other processors in order to request tasks. How the processors choose other processors to request tasks from varies and is determined on a per-problem basis.

2.2.3 Analytical modeling of parallel programs

A sequential algorithm is usually evaluated based on its execution time as a function of the input size. For parallel algorithms it is not so simple. They depend not only on input size, but also on the number of processing elements used (denoted by the symbol p), and their processing and communication speeds. In addition, a parallel algorithm cannot be evaluated isolated from the architecture on which it runs; a parallel algorithm designed to run on graphics processors will very likely not do very well on a standard multi-core computer. A *parallel system* is the combination of an algorithm and the parallel architecture on which it is implemented [15].

Performance metrics

We denote the serial runtime of a program by T_S and the parallel runtime by T_P.

Overhead The overhead that a parallel program incurs are combined into the *total overhead* and is defined as the total time collectively spent by the processing elements over and above that required by the sequential algorithm on a single processing element. We denote this by T_O. As the total time spent on solving a problem over all processing elements is pT_P and T_S time units are spent doing useful work, the remainder must be overhead. The overhead function T_O is defined as

$$T_O = pT_P - T_S. \tag{2.1}$$

Speedup *Speedup* is the measure that displays the benefit of solving a problem in parallel. It is defined as the ratio of the time taken to solve a problem on a single processing element to the time required to solve the same problem on p identical processing elements. We denote speedup by S.

Efficiency In an ideal world, a parallel system with p processing elements can give us a speedup equal to p. However, this is very rarely achieved. Usually some time is "wasted" by either idling or communicating. *Efficiency* is a measure of how much of the execution time a processing element is doing useful work given as a fraction of the time spent. We denote it by E and can define it as

$$E = \frac{S}{p}.$$

Scaling As shown the efficiency of a parallel program can be written as

$$E = \frac{S}{p} = \frac{T_S}{pT_P}$$

which can, by using equation 2.1, be rewritten as

$$E = \frac{1}{1 + \frac{T_0}{T_S}}. \tag{2.2}$$

We can see that T_O is an increasing function pf p. This is because all programs must contain some serial component. During this time all other processing elemtents must remain idle, therefore T_O must grow at least linearly with p. Due to idling, communication and possibly excess computation, this function in many cases grows superlinearly with the number of processing elements. Equation 2.1 shows us that if T_S remains constant (i.e. the problem size is fixed), as we increase the number of processing elements and T_O increases, the overall efficiency of our parallel program must go down. Decreasing efficiency with increasing numbers of processing elements for a given problem size is common to all parallel programs.

Amdahl's law

Amdahl's law [4] states that the maximum speedup that can be achieved is limited by the serial component of the program:

$$Maximum\,speedup = \frac{1}{T_S + \frac{T_P}{p}}$$

This means that for, as an example, a program in which 90% of the code can be made parallel but 10% must remain serial, the maximum achievable speedup is 9, even for an infinite number of processors.

2.2.4 Parallel random number generation

A Random Number Generator (RNG) (technically a pseudo-random number generator) is an algorithm that generates sequences of numbers that approximate the properties of truely random numbers, while being completely deterministic.

When designing a parallel program RNGs must taken into account. Most scientific computing applications, which are often ideal candidates for being parallelized, use random numbers for some purpose or another; especially for heavily RNG-reliant simulations, like Monte Carlo methods.

For some parallel programs, it is desireable that the stream is "shared" between processes, i.e. that the same stream (initialized with the same parameters) is used by each process, but each process skips a predetermined amount of steps in the stream for each number generated. However, for most programs it is more practical to seed the random number generator of each processs with their own, unique seed.

Much can (and has) been written about parallel random number generation; see for example [21], [8] and [20]. However, it was not within the scope of this thesis; therefore only one concern will be discussed, namely reproducibility.

Reproducibility

Reproducibility is desired for a number of reasons:

- Being able to reproduce results is vital to the scientific research method.
- Random numbers that are not reproducible makes writing tests hard.

Closely related to this is the issue of seeding. Random number generator library implementations usually seed their random number generators with the system time. A possible scenario is that several processes seed their random number generators with the same seed because of this. Two possible reasons: precision is bad and several processes start at the same time and thus get the same time value to seed with. With today's operating systems, this is not very likely to happen, as the precision is very good. A more likely scenario is that when running in a distributed manner, processes on different nodes could get the same seed if they start at roughly the same time (the system clocks on the different nodes are probably not perfectly synchronized).

2.2.5 Why write parallel programs?

When one sees the additional complexity that thinking about parallelism adds to the problem as compared to a sequential implementation, one might be dissuaded and instead opt for writing a sequential program and spend

more time on optimizing the algorithm, or perhaps simply invest in faster hardware. However, neither of these approaches are all that feasible. One of the the arguments for writing parallel programs that carry the most weight is the simple fact that it is no longer desirable (or even technically possible) to increase the clock speed of processors any more due to physical constraints (power and heat). And sequential algorithms can only be made so fast — at one point they simply cannot be made any faster.

2.3 Computer clusters

A computer cluster is a collection of computers where each computer runs under a separate instance of an operating system. Usually, these computers built from commercially available off-the-shelf parts and are connected to a very fast local area network via a fast interconnect. The computers used in today's computer clusters are typically MIMD architectures, i.e. each computer has several SIMD processors.

Computer clusters have several uses. One is load balancing, where several computers are linked together to serve as one virtual machine in order to distribute requests evenly over these computers. Another common use case is failover, or high-availability, clusters, where computers are linked together in order to provide improved availability for a service. If a component fails, a redundant, connected component within the cluster can seamlessly take over. Failover and load balancing are related and often overlapping concepts - load balancing is commonly used to implement failover. Both of these computer cluster types are commonly used to provide Internet services.

Of more interest to those in the field of scientific computing is the use of computer clusters for computational purposes. Connecting hundreds or thousands of MIMD computers together offers a huge amount of combined computational power. This allows for an effective way for both improving execution times and increasing the ability to handle larger datasets. Computer clusters that are designed with scientific computers in mind are often called *compute clusters*, as the focus is on computing. Compute clusters are typically built from off-the-shelf hardware, making them a cost-effective way of crunching numbers compared to traditional supercomputers.

Compute cluster control

Computer clusters are, in a loose sense of the word, managed by resource managers. These resource managers usually provides three key functions. First, it allocates access (be it exclusive or non-exclusive) to computer nodes (resources) to users so that they can perform work. Second, it provides a framework for starting and and monitoring work, typically a parallel job. Finally, it manages contention for resources by managing a job queue of pending work. The last two points are a necessity as computer clusters used for scientific computing are often shared between many researchers and research groups with varying interests. To provide the users with fair access the available computing power, good sharing policies and functionality are vital.

2.3.1 Titan

The Titan computing cluster is the high performance computing facilites at the University of Oslo. It consists of more than 650 nodes at the time of

writing, with over 5000 processor cores in total. The nodes are connected via Infiniband, a high speed interconnect.

Resource allocation with SLURM

The Titan computing cluster uses the computer cluster *resource manager* SLURM[5], an open source resource manager for Linux clusters, to handle job execution, resource allocation and arbitrate resource contention by managing a job queue. It supports tens of thousands of nodes and hundreds of thousands of processors and can handle a very high throughput of jobs. It is one of the most popular resource managers and is used by many of the world's most powerful supercomputers, such as the Chinese Tianhe-1A and the French Tera 100, respectively the world's and Europe's most powerful supercomputer, as of April 2011. To handle contention for resources, SLURM manages a *job queue* that governs when and for how long a job is allowed to run. This is implemented with *batch scripts*.

Batch script A *batch script* is a bash script[6] that describes a SLURM job. It contains special instructions to SLURM, as well as normal bash commands. Special parallel commands can also be run.

Listing 2.3: An example of a SLURM batch script. This contains the bare essentials for starting a serial job on a single node with a single core. Example taken from [7]

```
Line 1  #!/bin/bash
-
-       # Job name:
-       #SBATCH --job-name=YourJobname
5       #
-       # Project:
-       #SBATCH --account=YourProject
-       #
-       # Wall clock limit:
10      #SBATCH --time=hh:mm:ss
-       #
-       # Max memory usage:
-       #SBATCH --mem-per-cpu=Size
-       #
15      # Number of tasks(CPU cores):
-       #SBATCH --ntasks=NumTasks
-
-       ## Set up job environment
-       source /site/bin/jobsetup
```

[5] https://computing.llnl.gov/linux/slurm/

[6] a special kind of *shell script*, which is a script written for the command line interpreter of an operating system

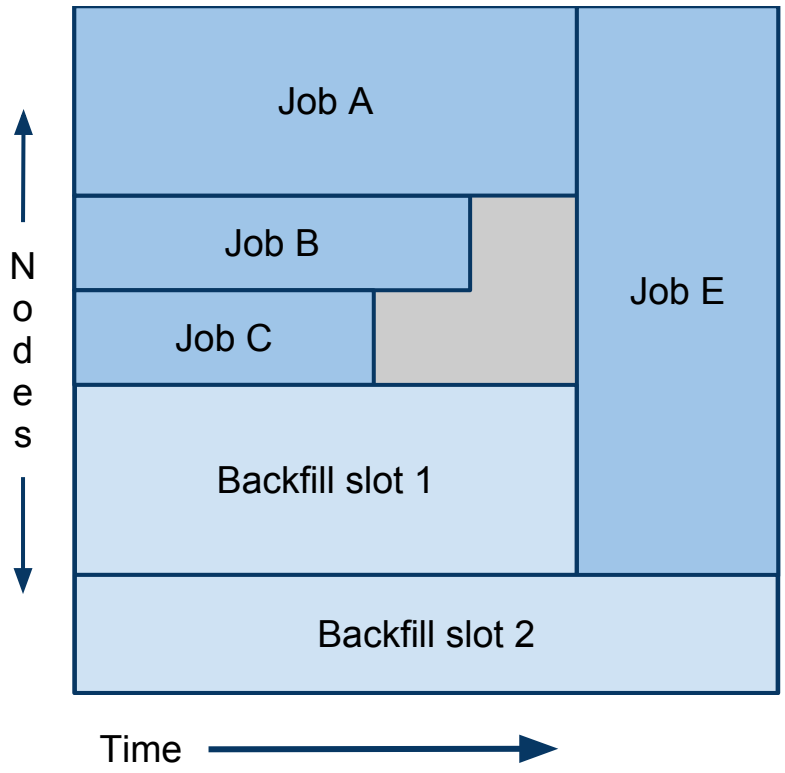


Figure 2.11: A demonstration of the backfill principle on a very small compute cluster.

```
20
- ## Do some work:
- YourCommand
```

As can be seen in the above listing, job length, memory use and number of cores has to be specified. These are the key characteristics that are used by SLURM to decide the job's place in the queue.

Job queue The SLURM job queue is, in theory, a First-In-First-Out (FIFO) queue. However, the queuing system is somewhat more complicated than that. In general, one can expect that the more system resources that is required in the batch script, the longer the job must wait in the queue.

The scheduling algorithm used on Titan can be thought of as a FIFO queue, but with some extra complexity. It can be thought of as having two axes; time along one and nodes occupied along the other. Thus, how much place a job takes in the queue can be thought of as a "box". These "boxes" are used to place jobs in the queue. While jobs of the same size will be executed in order, jobs of varying sizes may see the smaller job executed first, even if

submitted later than the larger job. This is because of the backfill principle: when a job finishes earlier than expected (e.g., a job that defined a 4 hour wallclock limit but only took 2 hours to complete), jobs in the queue may have their priorities increased so that they can run in the newly available slot. Figure 2.11 demonstrates the backfill principle. As can be seen in the figure, job E has reserved many nodes, and is therefore relegated to wait. New jobs that fit into backfill slot 1 or 2 will be allowed to run before it. Or, if jobs A, B and C finish before expected, job E may be moved up in the queue.

Because of this, having a "small" job may be preferable to a "large" job as small jobs gets more places in these backfill slots. While the actual runtime will be longer with the small job, the actual time from submission to completion might be shorter due to not spending as long in the queue. If the work that is to be done can easily be separated into independent parts, it is almost certainly faster to submit, for example, four one-hour jobs that each computes a quarter of the task at hand than a single four-hour job that computes everything at once.

2.3.2 Existing automated compute cluster functionality at the University of Oslo

The BioPortal[8] is a project at the University of Oslo that works in a way reminiscent of the system that has been developed for the HyperBrowser. It allows academic users to run many useful tools on the Titan computing cluster via a simple web interface. These tools are often complicated and have a command line interface; the BioPortal makes them more accessible for users who are not too familiar with such interfaces.

The framework presented here differs as the BioPortal is more similar to a collection of wrappers around command line tools that allow the use of these tools via a web interface than it is a unified system. It submits a job allocation to Titan for the user under a special BioPortal queue that has a set of reserved nodes, and when fulfilled it copies the user's own submitted data files to the allocated node(s), runs the requested command line tool, and copies the results back in file form. This saves the users from having to deal with complicated batch scripts and cryptic command line tools.

As such, it does not have altogether too much in common with the presented system. The HyperBrowser is most definitely not a command line tool. The only thing they really have in common is that they are both user-friendly interfaces to complex analytical tools, and that they both utilize the Titan computing cluster in order to analyze problems efficiently. Under the hood they are markedly different systems.

[8] `http://www.bioportal.uio.no`

2.4 Python

Python is a powerful dynamic, interpreted programming language that is used in a wide variety of applications. Some of its features include:

- clear, readable syntax
- exception-based error handling
- a very extensive standard library
- strong introspection functionality

For being an interpreted language, it is fast, and if speed is critical it easily interfaces with extensions written in faster languages like C or C++. A common way of using Python is using it for the high-level logic of a program, and for the portions of the programs where speed is critical, fast, low-level libraries written in pure C are used. As described in 2.1, this is the approach the HyperBrowser uses.

This section will present some technical details of Python that may aid in the understanding of the rest of this thesis. Do note that when talking about implementation specific Python topics, it can be assumed that it is the CPython implementation[9] that is meant.

2.4.1 Technical details

Python and threads

As mentioned briefly in Section 2.2.1, threads are the standard way of writing parallel programs. However, the Python interpreter is not fully thread-safe[10]. In order to support multi-threaded Python programs, a global lock, called the Global Interpreter Lock (GIL)[16], is used. This lock must be held by the executing thread before it can safely access Python objects. Without this lock, even the simplest operations could cause problems in a multi-threaded program.

Because of this, Python threads are not a viable choice for compute intensive applications, as it means that in practice only one Python thread in the same process can run Python bytecode at any given time. Processes are used instead, with independent interpreters in each process communicating through some form of Inter-Process Communication (IPC) channel, typically pipes.

[9] `http://www.python.org`

[10] `http://docs.python.org/c-api/init.html#thread-state-and-the-global-interpreter-lock`

Instance object creation

When a class type object (known as "new-style classes") in Python is called to create a new instance object, two methods are normally called: `__new__` and `__init__`. Usually, class type objects act as a factory for new instances of themselves. `__new__` is called first with the arguments of the call, which returns an instance of the class. This is then passed to `__init__` along with the arguments, where the instance members are set, superclasses called, etc. It it possible to override `__new__` in order to support customized instance creation; in this case, if `__new__` does *not* return an instance of the class, `__init__` is not called automatically.

Memory handling

Python features garbage collection, i.e. that memory management is handled automatically by the interpreter at runtime. Objects that are no longer in use by the program are automatically taken care of so that memory use is kept as low as possible. The automatic memory management in Python relies on two techniques to discern whether an object can be safely reclaimed or not: *reference counting* and *automatic garbage collection*.

Reference counting The concept of reference countingis fairly simple: every object has an associated counter, which is incremented when a reference to the object is stored somewhere and decremented when a reference to it is deleted. When this counter reaches zero, the object is reclaimed.

Automatic garbage collection Only relying on reference counting is not enough. Consider the three objects A, B and C, where A refers to B, B refers to C and C refers to B. If A's reference to B is removed, B and C will still have reference counts of more than zero, as they will refer to each other. This is known as a *circular reference*. In order to handle this, a cycle detector algorithm is periodically run by the Python runtime which finds inaccessible cycles and deletes the objects involved.

Weak references Occasionally references to objects are required, but one does not wish the references to keep the object from being garbage collected. This is commonly used for implementing caches, so that objects kept in the cache are automatically removed when they are no longer required, without requiring explicit removal. Python support this through *weak references*[11]. Weak references are references to objects that do not increase its reference count but can still be used like normal references. In short, this means that an object that only has weak references to it will be garbage collected.

[11] `http://docs.python.org/library/weakref.html`

Reflection

Reflection is the ability of a computer program to modify its own behaviour by introspection at runtime. Introspection can be defined as code looking at objects in memory, getting information about them, and manipulating them. In Python everything is objects, and more or less every detail of objects can easily be inspected at runtime. This allows for some interesting solutions to certain problems, but must be used with care. Otherwise it can easily lead to code that is hard to understand.

Interfaces

Python does not have interfaces as in Java or abstract classes as in C++. Instead, *protocols* are used to define something similar to interfaces. A protocol defines a set of methods that classes which wish to implement the protocol must support. This is not automatically enforced like Java interfaces. Instead, any object that "quacks like a duck" implements the "duck protocol" and is hence considered a duck (this is fittingly known as *duck typing*). This necessitates good programming practices so that it is easy to see what protocols a class supports. As interfaces are not defined in code, good documentation becomes important.

An example of this is *iterators*, which are used to iterate over the elements of a *container*. For an object to support the iterator protocol, all it has to do is implement the `__iter__` method (which is expected to return a *generator* object that supports the `next` method which returns the next element in the container).

NumPy

NumPy is a popular Python module for scientific computing. It offers a powerful N-dimensional array object, along with an easy-to-use array programming model. In the HyperBrowser, it is used for efficient operations on tracks.

Of special interest to us is the NumPy *memmap* functionality, short for memory mapped arrays. They are used for accessing small segments of large files on disk, to be able to read tracks without having to load entire files from the disk into memory at once.

Pickling

Pickle is a the standard Python module for serializing objects. Serialization refers to the process of converting an object to a format that can be safely sent over a network consection or stored in a file on disk. Serializing a Python object with the pickle module is referred to *pickling*. Most Python objects

can be pickled, but there are certain limitations[12]. Most notably, file objects cannot be pickled, nor can references to bound methods. Of interest to us is the fact that NumPy memmaps cannot be pickled either, due to using file objects internally.

Listings 2.4 and 2.5 demonstrate these issues.

Numpy Pickling memmap objects is not possible (see listing 2.4).

Listing 2.4: An example of how NumPy memmaps cannot be pickled. This example will raise an error on line 5.

```
Line 1  import numpy
    -   import pickle
    -
    -   mmap = numpy.memmap("test.map", dtype='float32',\
    5                        mode='w+', shape=(3,4))
    -   print pickle.loads(pickle.dumps(mmap))
```

Listing 2.5: An example of how the pickle module cannot serialize method references to other object's instance methods. This example will raise an error on line 13.

```
Line 1  import pickle
    -
    -   class A(object):
    -       pass
    5
    -   class B(object):
    -       def test(self):
    -           pass
    -
   10   a = A()
    -   b = B()
    -   a.test = b.test
    -   pickle.dumps(a)
```

Multiprocessing

The `multiprocessing`[13] module of the Python standard library is the standard way of implementing parallel execution of code in Python programs. It provides functionality for the spawning of subprocesses that can execute arbitrary methods and IPC functionality that can be used to communicate with these processes. It also provides locks and queues which can safely be communicated between processes and be used without worrying about

[12] `http://docs.python.org/library/pickle.html\`
`#what-can-be-pickled-and-unpickled`

[13] http://docs.python.org/library/multiprocessing.html

concurrency issues such as race conditions. In addition it has support for mapping embarassingly parallel problems to pools of worker processes automatically.

Unfortunately, it is also somewhat hard to use. Especially unhandled exceptions in child processes are handled in a manner that is less than satisfactory. An example of the inherent difficulty in developing parallel programs with the Python multiprocessing module can be seen in listing 2.6.

Listing 2.6: A code snippet that demonstrates the error handling issues of multiprocessing. This will hang forever in the parent process at line 11, as no `EOFError` will be raised (or indeed any exception) when the other end of the pipe crashes.

```
Line 1  import multiprocessing
    -
    -   def f(pipe):
    -       raise RuntimeError()
    5
    -   if __name__ == '__main__':
    -       localPipe, remotePipe = multiprocessing.Pipe()
    -       p = multiprocessing.Process(target=f, args=(remotePipe,))
    -       p.start()
   10       try:
    -           message = localPipe.recv_bytes()
    -       except EOFError:
    -           print "Other pipe end closed"
    -       except:
   15           print "Unhandled exception occured"
    -       p.join()
```

2.4.2 Python frameworks for map-reduce problems

As will be shown later, the HyperBrowser problems are for the most part embarassingly parallel, and is as such an ideal candidate for using the *map-reduce* programming model to efficienctly parallelizing it.

Map-reduce is a programming paradigm borrowed from functional programming languages. The *map* step splits a problem into smaller subproblems, while the *reduce* step collects the results from these subproblems and combines them to get the answer to the original problem.

Many frameworks exist for handling the map-reduce paradigm, like Apache's Hadoop (Formerly Google's MapReduce[9]), Microsoft's Dryad[18] and the Yahoo! Pig Latin programming language[23]. While these projects are interesting, they do not integrate very well with Python. An important point was the ability to integrate cleanly with the existing code; handling the parallelization with a framework in another programming language entirely would probably not have been a very successful endeavour.

Fortunately Python has many frameworks for parallel execution that are similar in concept, and in the following section some of the most well known will be presented.

A non-exhaustive list of Python frameworks/libraries for parallel execution

- Disco, a framework based on the map-reduce paradigm developed by the Nokia Research Center.[14]
- Execnet, a channel-based, easy-to-use framework for remotely or locally executing Python code in separate processes.[15]
- Celery, an asynchronous task queue framework based on distributed message passing.[16]
- Parallel Python, a simple, light framework for parallel execution of Python code both locally and remotely.
- Various MPI bindings/solutions, like pypar, pyMPI, mpi4py and Scientific.MPI.

In addition, there is a wealth of other libraries that were not considered, mainly because they are no longer being actively developed.

2.4.3 Parallel Python

Parallel Python (PP) is a Python module which allows for parallel execution of Python code on both Symmetric MultiProcessing (SMP) systems and computing clusters. It is released under a BSD-style license[17] and can be downloaded at `http://www.parallelpython.com`.

Its basic premise is that work, encapsulated in *tasks*, is sent to a *task server*, which then distributes the tasks evenly among *workers*. These tasks can be any callable Python object that can be serialized with the pickle module. The results are automatically collected when the tasks have been completed. Task servers can be distributed over many computers and act in a distributed manner.

Explanation of key Parallel Python components

Task server The task server is the central component of PP. Tasks are submitted to the task server, which stores them in a queue. From this

[14] http://discoproject.org/
[15] http://codespeak.net/execnet/
[16] http://celeryproject.org/
[17] A permissive free software license.

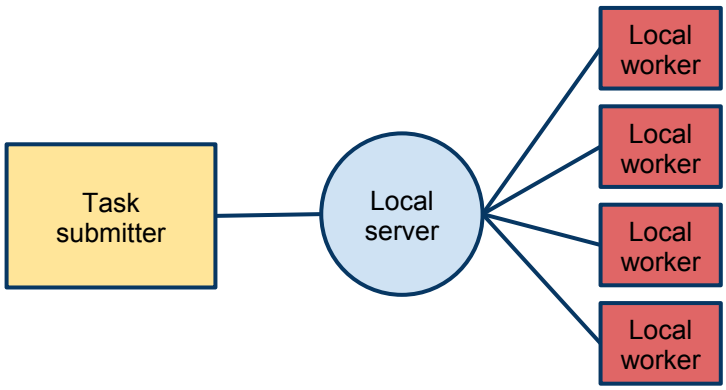


Figure 2.12: A simple example of a job running one local server managing four local workers

task queue tasks are distributed amongst all available workers. Workers can either be managed by the task server, or be connected to other task servers running on other machines. As there can be many connected task servers, it is important to separate between the *master* task server, to which tasks can be submitted, and *remote* task servers, which only receive tasks from the master task server and distribute these to their local workers. This closely resembles the hierarchical manager/worker task-scheduling algorithm described in Section 2.2.2.

Workers A worker in PP is a process spawned by a task server. They are connected to their task server via a pipe. Workers loop endlessly waiting for tasks from their task server. When they are assigned a task, they compute the result and return it to the task server.

Naming convention A worker can be either *local* or *remote*. A local worker is a worker that is managed by and receives tasks from the master task server. A remote worker is any worker which is not managed by the master task server, and are hence running on another computer and is managed by the task server running on that computer. This distinction is purely made to ease the understanding; a local and remote worker are exactly the same, implementation-wise. A worker does not care about where the tasks it is to execute comes from.

Task When a problem (in practice, a method that performs some computation) is submitted to the task server, a task object is returned. Technically this is a callable object that, when called, will block until the result is ready, or return immediatedly is the result has already been computed.

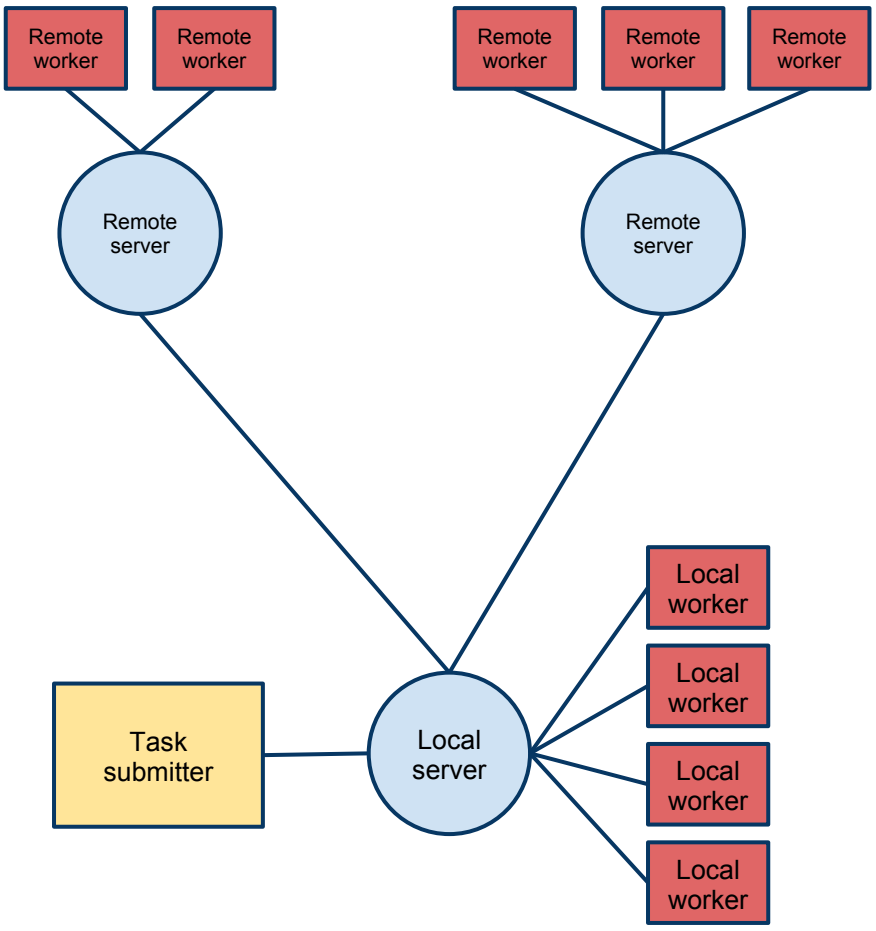


Figure 2.13: An example of a job running one local server managing four local workers working together with five remote workers spread over two remote servers

An interesting fact to note is that PP will actually transfer the actual *source code* of the method that a task comprises of. By using runtime introspection, the method's source code is sent to workers as a string, and then "compiled" and made runnable again on the worker side. This allows PP to run arbitrary code remotely; the workers do not have to have a local "install" of the method. PP uses caching of received methods on the workers so that once a task has been sent, any subsequent similar can used the cached method.

Parallel Python control flow

This section will explain how the various PP components interact and work together to compute the tasks.

The master task server, once created, will start up a predetermined amount of local workers, then connect to any predefined remote task servers. There is also support for automatically discovering and connecting to available remote task servers. Once the local workers have been started up, the task server is ready to receive tasks from the user and compute these.

The master task server maintains a FIFO queue that all submitted tasks are inserted into. As long as this queue is not empty, tasks are continously distributed amongst all available workers.

To keep track of what workers are available to process tasks, separate lists of local and remote workers are used. Local workers are put into the local worker list during initialization. Remote workers are added to and removed from the remote workers list as remote task servers are connected to/disconnected from. The workers are either considered free or busy; whenever a task is sent to a worker it is considered busy and it will not be sent any more tasks. When it returns the result of the task, it is again considered free.

The master task server chiefly spends its time in a scheduling loop. As long as the task queue is not empty, tasks are sent to any free workers. Local workers are preferred to remote workers if possible. Tasks are automatically retrieved and marked as ready once workers have computed the result.

Overlapping communication and computation

As explained in the previous section, two separate lists of local and remote workers are maintained. This is a simplification; in reality each remote worker is part of *two* remote worker lists, but as separate remote worker objects. Both of these lists are used when looking for an available worker; thus it is possible for a remote worker to be set to busy in one list, but be considered free in the other.

This is done to enable prefetching of tasks, as described in Section 2.2.2). Technically, as it is not the workers that request tasks, but rather the task

server that pushes them, preloading is better term. As remote workers are considered two "separate" remote workers, as long as one of them is considered free it can be sent a task (called a *reserved* task). Doing this allows computation of one task to happen while the other is being transmitted. Thus, when one task is finished, another task is already ready to be computed, without any delay.

Error handling

The ability to log jobs and their behaviour is important for all systems in case of errors. If a job crashes without any logging of the error, it obviously becomes hard (or impossible) to identify the reason for the crash. For parallel programs this can be an extra challenge. Luckily, PP has a fairly robust error handling system. Any unhandled error that occur during execution of a task will be caught by the worker. This error is then encapsulated (the traceback information is simply pickled) and returned to the task server, which can then return it to the job in lieu of a result. The job can then display the error to the user, just as if the error had happened locally. Only actual program execution errors will be returned in this manner; unforeseen events like network failure, etc. will be handled by the task server.

Auto-discovery of remote task servers

PP supports autodiscovery of remote task servers. When creating the master task server, one can either pass it a list of hostnames belonging to other computers that run task servers. One can also enable auto-discovery. If auto-discovery is enabled, *broadcasting*[18] is used to discover other task servers on the same network. This is particularly useful if it is not known beforehand where workers may be run at a later point in time. As long as they are run on a computer on the local network segment, they will automatically connect to the task server.

[18] broadcasting is a networking term for sending a message to a specific network address so that it reaches every computer on the same network segment

Chapter 3

Design

In this chapter a presentation of how the design principles presented in Section 2.2.2 was applied to the HyperBrowser, yielding the parallel implementation that is presented in Chapter 4. The Hyperbrowser was not written from the ground up by me and a major redesign was not really an option. Many of the design decisions had therefore in effect already been taken, which somewhat limited the options available at each step in the design process.

3.1 Design considerations

When designing the parallel implementation these were the design considerations that were the most important:

- Minimal change to existing code
- Ability to exploit both local and remote computing power
- Efficient handling of both large and small jobs

3.1.1 Minimal change to existing code

The Hyperbrowser is a large project that is still under active development. While the parallel solution was being developed, the development of the serial implementation was still continuing. An approach that had significantly changed the existing design would have made this concurrent development difficult. Because of this, minimal changes to the existing code base was important. If the parallel solution instead could be cleanly "dropped into" the existing solution, merging the two would be an easy task.

3.1.2 Ability to exploit both local and remote computing power

The University of Oslo has a large computing cluster available for which the research group the HyperBrowser is a part of has a sizable yearly allo-

cation. Today, this is not being fully utilized, while at the same time the HyperBrowser is expected to face significantly higher workloads as more researchers begin to use it. Being able to utilize this computing power would be useful, both as the Hyperbrowser is expected to serve more users in the future and as more advanced functionality with longer execution times, like the regulome functionality [25], is added to it. At the same time, the server that runs the Hyperbrowser is a fairly powerful computer in its own right. The ability to utilize both the computing cluster and the HyperBrowser server to cooperate in order to solve a large problem would be useful.

3.1.3 Efficient handling of both large and small jobs

The Hyperbrowser is an interactive system. Analysis parameters are chosen by the users. Therefore the "size" of the jobs, i.e. the runtime, can vary widely. The parallel system should be able to handle this.

Interactive jobs

The HyperBrowser jobs can vary greatly in size. As the users select the jobs parameters, of which there are a huge amount of possible combination, predicting the execution time of jobs is hard. Job runtimes can range from seconds to weeks. This is not a problem for the existing implementation; however, when dealing with compute clusters completely unknown execution times are problematic.

As explained in section 2.3.1, when submitting a job to a compute cluster, specifying a job execution time limit, also known as the wallclock limit, is required. This is due to how the SLURM scheduling algorithm works. If the execution time of the job exceeds this wallclock limit, the job is simply killed.

We are thus presented with a difficult problem — we do not know how much time a job is going to take, yet we are required to determine a time limit for it.

One approach to solving this problem would be annotating each and every analysis and track with a weight that somehow describes how "hard" or time-consuming this particular analysis or track is. However, this would be a huge undertaking. There is a very large amount of different data sets already present in the HyperBrowser database. Users can also supply their own data sets, or import them from external gene databases like UCSC[1] and Ensembl[2] or protein databases. In addition, there is constantly being generated enormous amounts of new data. Annotating all of these with "complexity" information is therefore an unsurmountable task.

[1] http://genome.ucsc.edu/
[2] http://www.ensembl.org/

3.2 Retroactively parallelizing the Hyperbrowser

When designing a program from the ground up, one can think of parallelization issues early, preferrably using a design strategy like PCAM to achieve a good parallel design. However, when parallelizing a system that has been designed without parallelism in mind is a quite different beast. The common thing to do is profile the code to see where the program spends the most time and then try and find a way of parallelizing these so-called "hot spots"[3].

As it turns out, the HyperBrowser handles problems that are inherently easy to parallelize — they can be said to be pleasingly parallel. Identifying the main hot spot is fairly easy. Cursory inspection of the code suggests that the `_doLocalAnalysis` method in the `StatJob` class (as shown in Figure 2.4, this occurs early in the job control flow) is a likely target for splitting the job:

Listing 3.1: `_doLocalAnalysis` from `StatJob` in the `StatRunner` module. `_userBinSource` is short for user defined bin source.

```
Line 1  def _doLocalAnalysis(self, results, stats):
    -           for region in self._userBinSource:
    -                   #...compute results for region,
    -                   #store in results argument...
    5           return stats
```

As can be seen in Listing 3.1, this method simply performs some computation for each bin (called regions in the code example) in series. As these bins are completely separate, this method stands out as a natural place to perform parallelization. In theory, the only limit on how much of a speedup one can expect from parallelizing here is the number of bins in the userbin. For a genome wide analysis with a bin size of 10 million, the number of bins is over 300. Profiling showed that almost all (over 99.9%) is spent inside this loop for most large runs.

3.2.1 Monte Carlo analyses

A special case exists for Monte Carlo analyses.

Monte Carlo analyses are always "managed" by instances of the `RandomizationManagerStat` class, as explained in Section 2.1.8. When the analysis is a Monte Carlo analysis, the most natural parallelization "spot" is moved from `_doLocalAnalysis` to the part in `RandomizationManagerStat._compute` where randomized tracks are created and results collected. See Figure 2.1.8.

Listing 3.2: Hot spot in `RandomizationManagerStat._compute`

```
Line 1  for i in xrange(self._numResamplings):
```

[3] A hot spot is a region of a computer program where a large portion of the execution time occurs

```
-           randChild = self._createRandomizedStat(i)
-           randResults.append(randChild.getResult())
```

From looking at Listing 4.1, we can hypothesize that the majority of the runtime is spent in this loop. Profiling of Monte Carlo analyses proved this hypothesis to be true. Moving the parallelization here is beneficial if we have perhaps only a small number of bins that we wish to analyze, but with many resamplings performed per bin. With the partition presented in the previous section, the maximum possible speedup would naturally be smaller, as it would be limited by the number of bins. As explained in Section 2.2.2, always use the partition which yields the largest amount of tasks if possible.

3.3 Applying theory

In the following section the design process is described. It follows the PCAM principle, as presented in Section 2.2.2.

3.3.1 Partitioning

Partitioning is the initial stage in the design process, where an initial decomposition of the problem into separate tasks as small as possible is performed.

In the Hyperbrowser, the most important data structure during the computation can be said to be the tracks. The decomposition of the data was more or less already decided from the outset; the Hyperbrowser works on tracks in bins, as shown above. The bins are completely separate from each other, and thus basing the partition on them is a logigal choice.

As explained in Section 2.1, for each bin a directed acyclic graph of child statistics is created in order to compute the result. A natural further decomposition of the problem would be to treat each of these statistics as a task. In theory these could be used as the basis for a recursive decomposition of the problem [15, pp. 95-97]. However, this would be difficult. For example, disk access is implemented through a statistic of its own; this alone would make data transfer between statistics cumbersome. In addition, the memoization scheme would have to be thrown away, which would likely make this partition perform poorly. Therefore, it was decided to rather think of entire bins as tasks, rather than individual statistic objects.

We can see that this partition yields a Single Process, Multiple Data (SPMD) problem; tasks are identical, they only differ in their input. This is ideal for parallelization.

One thing to note is that the tasks are not similar in size, i.e. runtime. As the size of the data that is worked on by each task is defined by the user, the actual data size (bin size in bp) is the same for all tasks, but the

amount of work required to actually process that data can differ because of the amount of data *points* varies between the regions.

As shown above, normal analyses and Monte Carlo analyses require different handling. Therefore, two partitions have been devised.

Normal analysis For a non-MC analysis, the natural place to perform the partition is in the local analysis phase of the job execution. Here, the program loops, doing work for each bin. Performing the partition here, with each bin being considered a task, seems like a natural place. The obvious drawback of this is that the maximum achieveable speedup is equal to the number of bins in the run. However, for most runs that take a considerable amount of time, there is a large amount of bins. For example, for genome-wide analyses with bins of 5mbp (a common bin size), there will be 600 bins.

Monte Carlo For Monte Carlo analyses, the natural place to perform the partition shifts. As shown in section 2.1.8, when Monte Carlo is to be used in an analysis, it is contained in a `RandomizationManagerStat`. The statistics that are to be performed many times (due to resampling) are "contained" in this statistic. The partition demonstrated in the previous paragraph will still work for Monte Carlo jobs, however it will not scale as the number of resamplings increase. Scalability is vital, therefore moving the partition to the `RandomizationManagerStat` for Monte Carlo analyses is done. This adds to the complexity, but the added performance makes it worthwhile.

Note that calling this a separate partition from the one mentioned above is actually somewhat misleading. The tasks are still similar, the only thing that differs is *where* these tasks are generated.

3.3.2 Communication

Using the partition obtained in the previous step, we can see that the communications requirements are very minimal. Both partitions in fact yield completely independent tasks; no inter-task communication is required at all. All that is required is supplying the tasks with initialization parameters, running them, and collect the results once they are done. No complex inter-task comunication is required. We can say that the decomposition yields a *pleasingly parallel* solution.

3.3.3 Agglomeration

In this step small tasks are combined to form larger tasks, in order to reduce the communication costs. As mentioned in the previous step, there is not much communication overhead.

Normal analysis For normal analyses an agglomeration step is not really necessary. For the most part tasks are "big" enough that agglomerating them is not required and simply reduces the possible flexibility.

Monte Carlo Monte Carlo problems, when partitioned with 1 sample per task, naturally produce a large amount of tasks. Because of this, implementing specific agglomeration strategies for Monte Carlo analyses is a good idea. Because MC tasks in essence consists of computing a statistic many times with a new randomized track each time, a special agglomeration scheme can be used for these. Simply mark every Monte Carlo task with a number n, and when ran, simply loop for n times and return a result list n long.

3.3.4 Mapping

Because the tasks are of varying size and the total number of processors is unknown at the time of the job creation (and can change during the execution of the job, due to how the compute cluster works), a static mapping would not have worked. Instead a dynamic mapping approach was chosen.

As the tasks that resulted from the partition scheme chosen were completely independent, the natural choice for the mapping algorithm was a worker/master based scheme. This offers good scaling, as well as being relatively easy to implement.

Chapter 4

Implementation

> The first 90 percent of the code accounts for the first 90 percent of the development time. The remaining 10 percent of the code accounts for the other 90 percent of the development time.
>
> *Tom Cargill, Bell Labs*

A framework has been implemented for distributing tasks across many computers, both locally and on a computer cluster, allowing efficient speedup of map/reduce problems. In the following chapter this framework will be described. Changes to the HyperBrowser that allows it to use this framework in order to gain significant speedups is also presented.

4.1 Implementation of design

The overall parallel design of the HyperBrowser was described in Chapter 3. As seen, it yielded a partition with completely independent tasks. On the basis of this, along with the other considerations presented, a general-purpose parallel framework has been designed which allows for concurrent users and the ability to exploit computing power both locally and remotely. The framework is naturally designed with the HyperBrowser in mind, and this shines through in many places. However, nothing directly links it to the HyperBrowser, and it could be used for other systems and problems as well.

It was decided to use an existing framework to handle the manager/-worker model and the distribution of tasks, for two main reasons. Firstly, an initial attempt at writing a new framework from the bottom up to do this showed that this would have taken too long (see Appendix B). Secondly, there is no need to reinvent the wheel simply to prove that you can.

When choosing what framework to use there were five main considerations.

Ability to work on both SMP systems as well as on clusters Being able to use the same framework for both an SMP system (namely the machine which runs the Hyperbrowser) as well as the computing cluster was a high priority. As getting an allocation in the computing cluster queue can take time, especially for larger jobs, being able to run jobs locally as well as on the computing cluster is useful. Another more practical issue, but possibly even more important, was that of development time. Even getting a five minute slot on a single CPU on the computing cluster can take hours if you are unlucky. Debugging an application with hours between runs is arduous and, quite frankly, nearly impossible.

Easy integration with existing code base Easy integration into the already large Python codebase was a requirement, as explained in Section 3.1.1.

Not require installation of anything on the compute cluster Installing background services or even just requiring an install on each compute cluster node was avoided as much as possible simply due to administrative concerns.

Automatic load balancing and error handling Automatic load balancing is obviously desireable; however why the ability to add computing power after starting a job is important may not be as readily apparent. The reason is the queueing system used on the compute cluster. As explained in Section 2.3.1, the compute cluster uses a queueing system. We do not have the privelege of having reserved remote computers on which we can run workers continously. Instead we must request computing power via the queueing system, and accept the fact that when we receive this computing power when we do. Therefore, the framework must be able to support workers "joining in" on the computation after it has begun.

The ability to add computing power on the fly Due to how the queueing system works, the available computing power had to be be able to be change over time; in effect adding or removing remote workers should be painless, or ideally automatic. As described in Section 2.3.1, the compute cluster queue favours short, small jobs. A system that can handle workers both leaving and joining often can exploit this — simply submit a large amount of small jobs in order to exploit the backfill window mechanic. Using short jobs also means that when the load drops off, the allocations will quickly die off, which will prevent wasted compute cluster resources.

Let us quickly review the frameworks or libraries presented in Section 2.4.2.

Disco is in extensive use and seems like it would be a good fit for the problems the HyperBrowser deals with. It works on SMP systems and clusters and has automatic load balancing. It even features practical extra functionality like a web interface to monitor nodes and has excellent performance. Unfortunately it requires the installation of custom software to work, and uses a specialized file system. In addition it does not seem to support adding more compute nodes on the fly.

Execnet features automatic bootstrapping and thus requires no manual installation. It lacks most of the other desired features, though. It does not have automatic load balancing and worse, it requires explicit host names to connect o the remote CPUs.

Celery uses an excellent shared task queue model which seems like a good fit, with automatic load balancing and workers automatically connecting when started without explicitly knowing their hostnames. However it requires a message broker to be installed on the computer cluster, and in addition does not seem to be designed for running workers on the local machine in addition to on remote machines.

The various MPI libraries mentioned do not fit the map/reduce model very well — they are more suited for programs in which frequent communication between tasks is required.

The framework that was chosen was instead Parallel Python, which was described in detail in Section 2.4.3. It has all of the desired features mentioned above. It has one key quality that made it favoured over the others.
The broadcasting functionality it provides allows workers to automatically connect to the master task server as they start up. This, coupled with the fact that the HyperBrowser server is located on the same network segment as the compute cluster, allows workers to be started up and shut down on remote machines throughout the lifetime of the master task server. Without this, the use of the compute cluster would have been much more difficult. The fact that it in addition is easy to use, features automatic load balancing as well as the ability to run workers locally made choosing PP an easy decision.

4.1.1 Overview

The system consists of three main parts:

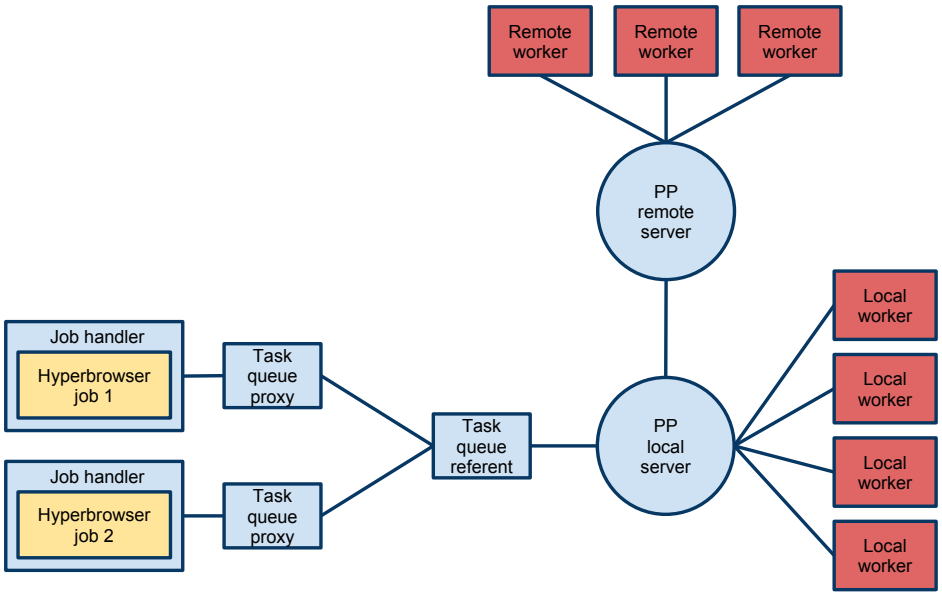


Figure 4.1: An overview of the components in the framework and how they are connected.

- The job handler, which manages splitting a job into tasks and combining the results
- The task queue, which manages concurrent access to a shared task queue
- Parallel Python, which handles the distribution of tasks to workers and gathering of the results

The key idea is based on the concept of reducing problems to tasks that can be computed independently in parallel. These tasks are submitted to a task server which manages a task queue. Tasks from this queue are automatically distributed amongst the available workers. These workers can be running both on the host computer as well as on other, remote computers. The task server can concurrently handle more than one problem, in order to be able to handle more than one user simultaneously.

The overall flow is easily seen from Figure 4.1. Problems are wrapped by job handlers, which split the problem into independent tasks. These tasks are then submitted to a task queue, which Parallel Python then uses to distribute the tasks amongst its workers.

4.1.2 Job handler

A *job handler* act as an entry point for jobs to the parallel framework. To the job handler one submits parallelizable jobs through the use of *job wrappers*, which can be thought as a kind of decorator that decorates jobs with functionality that allows it to be computed in parallel.

Job wrappers are used to describe how a problem is to be solved by the framework. It is implemented like a Python protocol that allows the framework to split the problem into separate tasks. These tasks must be picklable (see section 2.4.1), so that they may be communicated to the workers.

Making a problem implement the job wrapper protocol is done in a fairly simple, yet perhaps somewhat untraditional way: In a Python module, two classes are defined; one that wraps the problem, and one that wraps the task. The problem wrapper is then passed to the job handler, which takes care of the necessary setup before computing the problem in parallel and returning the results (see appendix A for a code example of how to do this is done).

Rationale

The rationale for this design is a consequence of the design of the HyperBrowser. In particular, it is a consequence of the memoization scheme the HyperBrowser uses, as well as the difficulty in pickling many of the objects used.

HyperBrowser memoization The HyperBrowser uses a memoization scheme in order to avoid recomputation of results as explained in Section 2.1.7. However, other parts of the system depend on the memoization in a manner that may not be immediately apparent to the observer. Mainly, it is the global analysis phase that occurs after the local analysis phase that depends on the results being present in the memoization table.

In the global analysis phase, the memoized results from the local analysis are used to compute a global result very quickly, as all the local results (the results per bin) are stored in memory. This is a sound design when the computation happens locally so that the memoization lookup table contains every computed result after the local analysis phase. However, when the local analysis phase happens remotely, the memoization cache will be empty. Therefore, some way of populating the memoization table with the results from the local analysis that is carried out remotely was needed, so that the remote computation would be completely transparent to the rest of the system.

After a task has been computed and the results are ready, some handling of the results is probably required before they can be returned to the rest of the system. As a chief concern is being able to simply drop the framework into an existing code base to replace a function call, the rest of the code

should not have to be changed at all. An easily imagined scenario is that the workers perform some work on a large matrix and return submatrices as results that have to be combined before being returned to the rest of the system.

For the HyperBrowser, having to process the results from the workers happens when returning the results from a non-Monte Carlo (MC) job.

In order to do this, a way of serializing the memoization lookup table on the worker was required. After a bin has been computed, the results are extracted from the lookup table (in `MagicStatFactory`, presented in section 2.1.7). Both in order to avoid issues with the pickle module (in case the statistic contains an object that is not picklable) and in order to reduce the size of the serialized object, a dictionary with `StatisticResultHolder` objects is created. These objects are dummy objects that implement all the methods that are expected from a `Statistic` object (the statistic protocol), but in actuality does nothing but hold a result. When the results are returned from the workers, they are simply dictionaries containing these objects. These dictionaries are combined in order to create a full dictionary that can simply be inserted into the `MagicStatFactory`. When the global analysis is performed, these objects contain the necessary results and it is near instant.

Hide away complexity A goal was to minimize the amount of code that was added to the exitsting HyperBrowser files, and this seems to achieve it in a fairly good manner, hiding the complexity of splitting the job away in the job wrapper specifications. For example, this is all the code that is required to start parallel Monte Carlo computation:

Listing 4.1: From RandomizationManagerStat.py

```
Line 1  jobWrapper = RandomizationManagerStatJobWrapper(self,\
    -                              seed=self._kwArgs["uniqueId"])
    -  jobHandler = JobHandler(self._kwArgs["uniqueId"], \
    -                          useSharedTaskQueue, True)
    5  randResults = jobHandler.run(jobWrapper)
```

4.2 Task queue

The HyperBrowser is not a single-user system; it is designed to allow many concurrent users. Because of this, as explained in Section 2.1.1, Galaxy spawns a new process for each job a user submits, so that jobs are completely independent of each other. Thus, in order to make all currently running jobs be able to utilize the same task manager, some way of sharing a Python object is required.

Python does have support for shared memory objects[1]. However, it is not very good. It only supports ctypes objects, which are both hard to write and use. In addition, even if a normal Python object could easily have been made a shared object, there is still the issue of how to communicate this shared object to each job. The common approach with shared objects is to pass a reference to the shared object around to everything which is to use said shared object. For this to be possible with the current Hyperbrowser design, modifying the Galaxy code would be necessary as it would have to manage the shared objects so it could pass them to jobs as they were created. Modifiying as little as possible of the existing code base was a chief concern, so this approach was not desirable.

Luckily there is another way of sharing objects provided by the multiprocessing package, namely *managers*[2]. Managers provide a way to create and manage objects which can be shared between different processes in an easy-to-use manner. A manager object controls a server process which contains shared objects. Other processes can access the shared objects by using proxies. Proxies are local objects that are used to access the corresponding remote object; this remote object that the proxy *refers* to is called its *referent*. The remote object resides in another process; this process can either be on the same machine as the proxy or on another machine entirely. The proxy and its referent communicate in the same manner regardless of whether the proxy and its referent both run on the same machine. When methods are called on proxy objects, the method call is simply forwarded to the referent, which executes the method call and returns the result to the proxy. Return values and arguments are serialized before being communicated, like everything else in Python that is involved in interprocess communication.

The manager/proxy solution is fairly technical and perhaps hard to understand. The short explanation is that it allows multiple processes to use a shared object. Using a manager that one communicates with via the network requires no such modifications to the Galaxy code base. Using the network for communication simplifies thing as new jobs simply try and connect to any existing manager without knowing anything more than which network port to search for the master task server on.

Another useful thing with this approach is error handling. If the job manager for some reason crashes or has to be restarted, any running jobs can detect that the job manager has disconnected and try to reconnect. If the shared object has been passed in as a reference, this would have been impossible unless the passed in shared object was actually a wrapper around the shared object that supports error handling, adding to the complexity.

In the implementation presented here, this is used to provide shared

[1] `http://docs.python.org/library/multiprocessing.html#shared-ctypes-objects`

[2] `http://docs.python.org/library/multiprocessing.html#module-multiprocessing.managers`

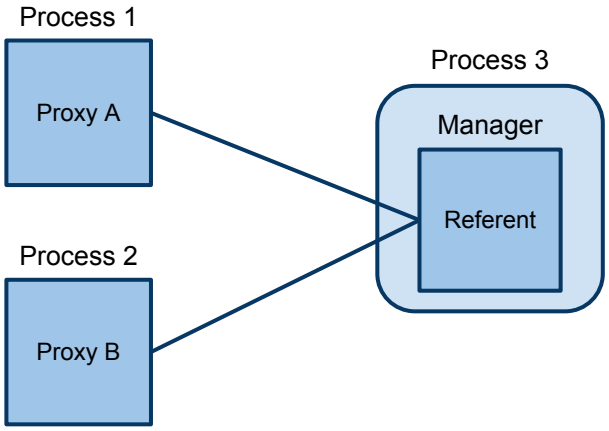


Figure 4.2: An example of one manager managing a single shared object with two proxies, all living in separate processes. The lines between the proxies and their referent signifies an open interprocess communication channel, typically a pipe.

access to the master task server, so that concurrent jobs can submit tasks to it simultanously.

A task queue manager `TaskQueueManager` runs as a background service. The manager objects controls access to a `TaskQueueReferent` which runs in a separate process. Jobs connect to this manager via the local network interface. The manager return proxy objects (`TaskQueueProxy`) to the jobs, which then use these to submits jobs to the shared task queue. For the jobs, this happens completely transparently; a `TaskQueueFactory` is used to create an object which implements the task queue "interface" by the `JobHandler`. This can either be a `TaskQueueProxy` (the default, used to access the shared queue), or a `LocalTaskQueue`, if a separate task queue is for some reason required for the job.

4.3 Parallel Python

The design and inner workings of Parallel Python was described in detail in Section 2.4.3 and will not be explained again here.

However, while Parallel Pyhon for the most part performs very well, some changes were made to it in order to achieve the desired performance, and these will be described.

Spawning of workers

Originally workers were spawned by the subprocess module[3]. However, this led to jobs taking much longer to finish than the original implementation; test runs with a single local worker were roughly 7-8 times slower than the original implementation. Profiling of these runs was performed to uncover why this happened. This showed that NumPy calls were taking much more time than they did when compared with the single-threaded original implementation. Uncovering the reason for this happening was not successful. The problem was fixed by switching to using the multiprocessing module to spawn the workers. This should be investigated further, see Section 7.7.2.

The original implementation communicated with workers via file-like objects (returned by the `subprocess.popen` call which spawns the process). This was implemented in the `PipeTransport` class in `pptransport` and used these file-like objects (pipes) to send and receive directly from the worker's `stdin` and `stdout` streams. Processes made with the multiprocessing module cannot be communicated with in the same way, as when spawning new processes with the `multiprocessing` module, pipes to the process' `stdin` and `stdout` streams are not provided. Because of this, a new class (`MultiprocessingPipeTransport`) was written, which uses a `multiprocessing.Consection` object to perform the communication instead. Instead of using separate one-way "send" and "receive" file-like objects, it uses a duplex pipe (`multiprocessing.Pipe`). As a result, the class constructor signature for the `_Worker` class in `ppworker` had to be changed slightly so that it accepts a pipe object. With the old implementation, workers simply communicated with their task server by reading from their `stdin` stream or writing to their `stdout` stream. As explained this does not work for processes created with the multiprocessing module. Instead, a pipe object is passed to the child process during its initialization that it uses to communicate with the parent process.

In addition, a small helper class (_WorkerProcessRunner) was created, simply to launch the worker process. With the subprocess module, a worker process takes an argument like "python worker.py" and executes this command in a newly spawned subprocess, which makes the `__main__` method [4] of the process the starting point. With multiprocessing, a method that is to be executed in the child process is passed to it instead.

[3] `http://docs.python.org/library/subprocess.html`

[4] Python does not actually have a main method like other languages like Java or C, instead common practice is to perform the check `if __name__ == '__main__'` at the bottom of the program and execute the "main" code there

4.4 Compute cluster functionality

One key point which has not yet been adressed is how compute power is requested from the compute cluster described. It is somewhat separate from the rest of the framework.

4.4.1 Reserving computing power on the computer cluster

As explained in section 2.3.1, the Titan computer cluster uses the SLURM resource manager. Jobs are submitted to SLURM via batch scripts.

The submission and writing of these batch scripts which describe the job is usually a rather manual job. Functionality has been implemented to make the submission of jobs a more automated process. It allows use of the computing cluster without having to manually write bash scripts when compute cluster time is required.

Batch script generation and submission

Whenever more computing power is required, a description of the job as needed by the batch system is required, a batch script has to be generated and submitted. This has been implemented in `SlurmBatchScriptBuilder` and `SlurmWrapper`. These classes act as an interface through which the computer cluster can be utilized by other functions; they mask away the SLURM batch script implementation details.

When asking for more computing power, if nothing else is specified, it uses default values of reserving entire 8-core nodes with 16GB of memory for timeslots of one hour. The command that is to be run when a job allocation is received is the PP remote task server. The job is then submitted to the computing cluster's job queue with the `sbatch` tool.

Once the job receives an allocation, a PP remote task server starts up and spawns one worker per core on the allocated node. The remote task server then locates the master task server, as explained in section 2.4.3, and connects to it. The remote workers are then ready to perform work.

Automatic job allocation

When requesting more computing power in the manner described above, it can be done manually by the way of a Python script (run from the command line). In addition, a background srveice has also been implemented (TitanJobAllocator.py), which if ran continously monitors the shared task queue for the presence of tasks and requests more computing nodes if deemed necessary. Currently this is mostly a proof of concept; it simply inspects the queue every n minutes and requests more workers on Titan if it is not empty.

4.4.2 Special compute cluster considerations

When running programs on a computer cluster, some extra considerations must be taken. As shown in listing 2.3, the memory requirement per processor core must be specified. This is a limit which is harshly enforced by SLURM; if the memory limit listed in the batch script is exceeded, the job is immediately killed.

Ensuring that this memory limit is not exceeded can be somewhat difficult with a language like Python where the memory use is seldom a concern (due to automatic garbage colsection) and usually not explicitly handled by the programmer. Luckily, Python offers the `resource` module [5].

The resource module offers the ability to register resource limits. This is used on the remote workers to ensure that memory use is kept within the specified limits. Currently, as only entire nodes are allocated, this has been disabled as it is not necessary (the workers per node collectively "own" the entirety of the memory on the node, as such the quota is not enforced).

4.5 Reproducibility and random numbers

Being able to reproduce experimental results is an essential principle of the scientific method. A recent investigation found that less than half of 18 microarray experiments published in Nature Genetics in 2005 and 2006 could be satisfactorily reproduced [17]. Naturally, we should strive for a system where every result is verifiable. For this to be possible, every parameter that has any bearing on the final result must be stored. "Randomness" and "reproducibility" are not exactly words that go together very well. If the numbers are truely random, reproducing the result is impossible. Luckily, most "random" numbers used in simulations (like MC models) are not truely random, they are instead generated by pseudo-random, deterministic number generators. Provided the initial value used to seed the number generator is stored, reproducing the results is possible; simply use the same seed to rerun the simulation and the same sequence of "random" numbers will be generated.

However, when the program is not sequential, this problem is not quite so easily solved.

An example from the HyperBrowser is finding out whether segments from one track overlap segments from another track more than expected than chance. A MC approach is used to randomize the positions of the segments. To reshuffle the segments, NumPy is used. NumPy by default uses random numbers supplied by the operating system if possible (/dev/urandom on Unix-like systems) if possible, or the system clock if not, to seed its RNG. This does not lend itself very well to reproducibility.

[5] `http://docs.python.org/library/resource.html`

To solve this, a way of supporting user-provided seeds has been implemented. For MC runs, when a task is prepared for parallel execution, a predefined seed can be supplied (however, do note that currently no support for this has been added to the web interface). This user supplied seed is combined with enough unique identifiers (sample number and bin number) so that each task has its own unique seed, which is used to seed the RNG on the workers before executing each task. This gives reproducible results.

This is implemented only as a proof of concept; no guarantee for the validity of the number theory behind this is given. See section 7.2 for suggestions regarding future work.

4.6 Using the framework in the HyperBrowser

4.7 In the HyperBrowser

When using the framework in the HyperBrowser, recall the design as described in Chapter 3. Let us do a quick reap: For a non-MC job, parallelization occurs in the `StatRunner.run` method. For a MC job, it instead occurs in the `compute` method of `RandomizationManagerStat`.

`StatRunner.run` is the "entry point" for all calculations. It is here that both the local and global analysis phases are started.

`RandomizationManagerStat` is a statistic which implements MC resampling in the HyperBrowser. This is done on a per bin basis.

Because of this this, where the choke point is during an analysis differs between the two types of jobs. In a "normal" analysis, the natural place to perform the partitioning is in doLocalAnalysis in StatRunner. However, for MC jobs the choke point moves to the `_compute` method of `RandomizationManagerStat`.

To detect whether an analysis is a MC analysis before splitting it into tasks, the statistic "tree" (described in Section 2.1.6) is constructed at the host computer, but without performing any computation; the statistics are simply instantiated. The tree is then traversed. If `RandomizationManagerStat` is found in the statistic tree, the MC parallelization scheme is used.

To do the actual splitting of jobs into disjoint tasks, job wrappers are used, as described in the framework description. Different job wrappers are used for MC jobs and non-MC jobs. The reason for this is, as described above in Section 4.1.2, the `MagicStatFactory`. For non-MC jobs this must be returned in a serialized form; hence it needs its wrapper to handle this. MC jobs, on the other hand, only return a list of numbers, therefore it has a much simpler wrapper.

In addition to defining the different ways to handle the returned results, the wrappers also describe how to split the job into tasks. This is in theory fairly simple — simply make each task a statistic.

However, in order to send Python objects to another process, be it over a pipe or a network connection, the object must be picklable, i.e. it must be serializable. However, certain objects are not picklable, such as NumPy memmaps or open Python file objects. Statistics contain memmaps.

To solve this and various other problems, a way of serializing statistic objects had to be implemented. For statistic objects, this in effect means storing everything needed to reconstruct the object in another process as strings in an object so that this object can be pickled (see Section 2.4.1) and transferred.

4.7.1 Making tasks picklable

Each task consists of a statistic that is to be performed. However, a statistic object is often not picklable, due to tracks (which contain NumPy memmaps), bound methods, etc. Also, for example for MC analyses, one would not be interested in instantiating tens of thousands (or even millions) of objects on the server machine, only to pickle them and then transfer them to workers. It is much faster to simply pickle the statistic initialization parameters, and then use these to create the "same" object on the worker.

In order to aid in the pickling of statistics, necessary helper classes have been implemented in `PickleTools`. These can be used to create objects that encapsulate the parameters required to create a statistic object. This mainly consists of making the tracks picklable. This has to be done as the track used in the computation are made during initialiazation of the HyperBrowser job, and they contain memmaps that make them not directly picklable. Therefore, the necessary arguments are "extracted" from it through introssection at runtime, such as track name, track class and format converters[6]. Different handling of different types of tracks is required. Currently two types of tracks are handled; `RandomizedTrack` (used in in MC analyses) and the "standard" `Track`. They are fairly similar, the only real difference is that randomized tracks are defined by an "original" track and a "randomization index". The original track is randomized based on this randomization index during creation of the randomized track.

4.8 Overview of code

No code has yet been shown. This is because is is the overall design that is interesting — the code itself is not all that important. Regardless, here

[6]Format converters are used to automatically convert data from the tracks to the type required by the statistic at runtime. For example, segments can be converted to points by using the median value.

follows a description of the where in the code the various components described in this chapter can be found. My code is mostly placed in the `quick.application.parallel` module, with necessary HyperBrowser changes in the `gold.statistic` and `quick.application` modules and Parallel Python changes in `third_party.pp`

See Appendix C for instructions on how to view the source code.

File name	Contains
Own files	
quick.application.parallel	
ClusterConfig.py	Configuration options for compute cluster, such as number of processors per node and maximum memory use.
ClusterSetup.py	Functionality for limiting resource use for workers running on the compute cluster to prevent being killed by the resource manager.
Config.py	Configuration options for the parallel implementation. For example, the port which Parallel Python uses, along with the passphrase it uses to authenticate workers.
JobHandler.py	Job handling functionality described in Section 4.1.2.
JobWrapper.py	Job wrapper functionality, as described in 4.1.2.
PickleTools.py	Functionality for pickling statistics and tracks, as described in Section 4.7.1.
StartTitanJob.py	Script for submitting job allocations manually.
TaskBatch.py	Functionality for agglomerating tasks, used by MC tasks.
TaskQueue.py	Implements the task queue, as described in Section 4.2.
TaskQueueManager-Referent.py	Script which runs the task queue manager. Must be run so that several jobs can connect to the same queue.
Titan.py	Titan-specific functionality. For example SLURM specific functions for submitting tasks.
TitanJobAllocator.py	Automatic compute cluster allocation functionality background service. Mostly a proof of concept.
Worker.py	Worker specific functionality. Describes what a worker is to do once started by Parallel Python.
HyperBrowser files	
quick.application	
GalaxyInterface.py	Minor changes to extract the unique job number from Galaxy to use as an identificator in the queue.
StatJob.py	Code to perform non-MC analyses in parallel.
gold.statistic	
Randomization-ManagerStat.py	Code to perform Monte Carlo analyses in parallel.
Parallel Python	
third_party.pp	
pptransport.py	Changes to allow PP workers to use `multiprocessing.Pipe` objects for communication.
ppworker.py	Added functionality for profiling worker processes, as well as some extra bootstrapping functionality.

Chapter 5

Results

For the following results, local workers (workers that run on the Hyperbrowser machine) were limited to 24. Any more workers could have led to the results being affected by other work being done on the machine at the same time, as the machine was in use by others for other kinds of work at the same time.

5.1 Hardware setup

Hyperbrowser server machine The Hyperbrowser server machine has four octocore (for a total of 32 cores) Intel X7550 processors running at 2ghz and 160 gigabytes of memory. Data is stored locally on normal SCSI hard drives via a SATA interface.

Compute cluster On the compute cluster Titan, the hardware varies. Most nodes are Sun X2200 blade nodes with two quadcore (for a total of 8 cores) AMD Opteron processors running at 2.3ghz with 16GB of memory. These are in the process of being switched out for Intel-based blades with 24-32 cores and 32-48GB of memory. It is very much an heterogenous environment.

For analyses where workers have been run on the compute cluster, have been limited to nodes with the same hardware (X2200 blades as mentioned above) for all runs. As the cluster used is heterogenous, allowing the jobs to run on arbitrary nodes could easily result in inconsistent results. For runs with less than eight workers, entire nodes have been reserved and fewer than the reserved number of cores have been used, in order to avoid having other users' jobs run on the same node.

The data used is stored on a distributed file system called the IBM General Parallel File System (GPFS) [1]. GPFS is designed for high performance, parallel access to petabytes of data that spans hundreds or thousands of

[1] `http://www-03.ibm.com/systems/software/gpfs/`

disks. Compute cluster nodes are connected to the disk nodes via Infiniband [2], a very fast, scalable interconnect.

For all tests, the results are averaged over at least five runs, in most cases more. More runs would have been preferrable, but time limitations due to having to wait in the compute cluster queue for sometimes 12 or more hours made this difficult.

5.2 Framework test results

The summation example from section A.1 is used in order to demonstrate the framework itself, isolated from HyperBrowser specifics.

t is the number of tasks executed, while n is the amount of numbers summed per task. For $t = 100, n = 100$, for example, the numbers from 0–100 are summed 100 times (once for each task). This is of course simply an example to demonstrate performance; the actual computation is pointless in itself.

For reference, the single-threaded run , the exact same code is run, except rather than using a JobHandler to run the code in parallel, the code is run in serial. I.e., rather than doing this:

```
Line 1      jobHandler = quick.application.parallel.Sum.\
-                   JobHandler("dummyId", useSharedTaskQueue=True)
-           result = jobHandler.run(wrapper)
```

, we exploit that wrappers are iterators and do this to run the same code in serial:

```
Line 1      taskWrapper = SumTaskWrapper()
-           for task in wrapper:
-               taskWrapper.handleTask(task)
```

5.3 HyperBrowser analysis results

Artificial results are not too interesting. Let us look at some representative results from the parallel implementation of the HyperBrowser.

5.3.1 Usage scenario 1: Histone modifications vs. SINE repeats

Usage scenario 1 is a Monte Carlo analysis of H3K27me3 histone modifications (track 1) and SINE repeats (track 2) in the mouse genome. The hypothesis is "are MEFB1 (BLOC segments) overlapping 'SINE (Repeating elements)', more than expected by chance?". It is the analysis used in [24].

In each bin, the test of

[2] http://en.wikipedia.org/wiki/InfiniBand

Local workers	$t = 10^5$, $n = 10$	$t = 10^5$, $n = 10^5$	$t = 10^5$, $n = 10^6$
1	104.5	169	1479
2	104.7	105	738.4
4	104.9	104.9	367.7
8	104.7	104.6	184.2
16	105	105.1	103.5

(a) Runtimes in seconds using local workers.

Local workers	$t = 10^5$, $n = 10^6$	$t = 10^4$, $n = 10^7$	t = 10^3, $n = 10^8$
1	1479	1393.4	1382.2
2	738.4	693.1	688.7
4	367.7	345.9	345
8	184.2	172.7	173.9
16	103.5	88.3	89.3

(b) Runtimes in seconds using local workers. Note that the overall computational complexity of all of these tasks are the same; they all equate to 10^{11} additions. They only differ in the number of tasks these are spread over.

Remote workers	$t = 10^5$, $n = 10$	$t = 10^5$, $n = 10^5$	$t = 10^5$, $n = 10^6$
16	257	263	318
32	129	130	160
64	103.9	103.5	104
128	103.8	103.3	103

(c) Runtimes in seconds using remote workers.

Remote workers	$t = 10^5$, $n = 10^6$	$t = 10^4$, $n = 10^7$	t = 10^3, $n = 10^8$
16	318	128.7	128.3
32	160	65	65.4
64	104	30.7	33.2
128	103	16.37	17.26

(d) Runtimes in seconds using remote workers. Note that the overall computational complexity of all of these tasks are the same; they all equate to 10^{11} additions. They only differ in the number of tasks these are spread over.

Figure 5.1: Results from example on how to use the framework. Demonstrates a fully CPU bound problem.

Workers	Local workers	Workers on compute cluster
1	1.08	1.02
2	2.15	2.02
4	4.28	3.76
8	8.45	6.05
16	15.53	11.92
24	21.52	17.61
32		21.14
64		36.04
128		61.9

(a) Speedup compared to the original implementation.

Workers	Local workers	Workers on compute cluster
1	1	1
2	1.99	1.98
4	3.96	3.69
8	7.82	5.93
16	14.38	11.69
24	19.93	17.26
32		20.73
64		35.33
128		60.69

(b) Scaled speedup. Both columns have been scaled by the speedup achieved by a single worker.

Figure 5.2

H_0: The segments of track 1 are located independently of the segments of track 2 with respect to overlap

vs

H_1: The segments of track 1 tend to overlap the segments of track 2

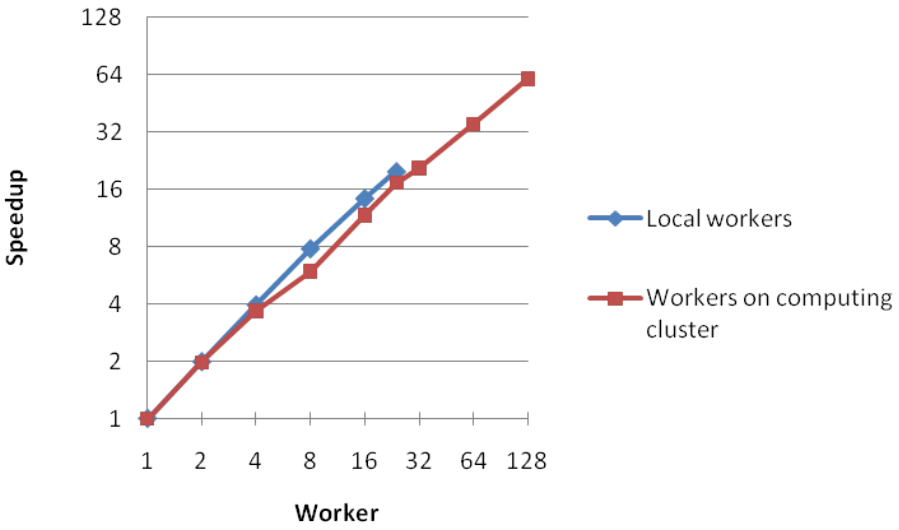
was performed.

P-values were computed under the null model defined by this preservation and randomization rule: Preserve segments of track 2, segment and inter-segment lengths of track 2, randomize positions (by Monte Carlo).

The test statistic used is “The number of base pairs that are inside segments of both tracks“ The amount of resamples combined into a single task was set to 100. The genome used is mm8 and the bin size is 5mbp.

Workers	Local workers	Workers on compute cluster
1	1	1
2	1	0.99
4	0.99	0.92
8	0.98	0.74
16	0.9	0.73
24	0.83	0.72
32		0.65
64		0.55
128		0.47

(c) Efficiency. Data used is from (b).



(d) Plotting the data in (b). Note the use of the logarithmic scale on both axes.

Figure 5.2: Speedup versus number of workers for Usage scenario 1: Histone modifications vs. SINE repeats, both with workers running on the HyperBrowser machine and on the compute cluster. *20000* resamples. For reference, the single-threaded implementation uses 717 seconds on this analysis.

Workers	Local workers	Workers on compute cluster
1	1.95	1.11
2	3.85	2.19
4	7.6	3.39
8	14.64	6.18
16	25.29	10.95
24	34.78	15.99
32		21.04
64		38.11
128		65.31

(a) Speedup compared to the original implementation.

Workers	Local workers	Workers on compute cluster
1	1	1
2	1.97	1.97
4	3.9	3.05
8	7.51	5.57
16	12.97	9.86
24	17.84	14.41
32		18.95
64		34.33
128		58.84

(b) Scaled speedup. Both columns have been scaled by the speedup achieved by a single worker.

Figure 5.3

5.3.2 Usage scenario 2: TFs vs. diseases

Usage scenario 2 is a non-MC analysis. It gives a descriptive statistic of ”A few TFs“ (track 1) versus “A few diseases” (track 2). Both of these tracks are, as the name suggests, sample data mainly used for testing. This analysis in itself is not altogether too useful, it is mostly used as a good example as it takes a decent amount of time, handles large datasets and is divided into many relatively small bins. In addition, it is the most time consuming part of the regulome functionality the Hyperbrowser provides [25].

The analysis can be formulated as ”What is the number of “A few TFs“ inside ’A few diseases’, for all combination of categories from both tracks” and is performed over the entire genome.

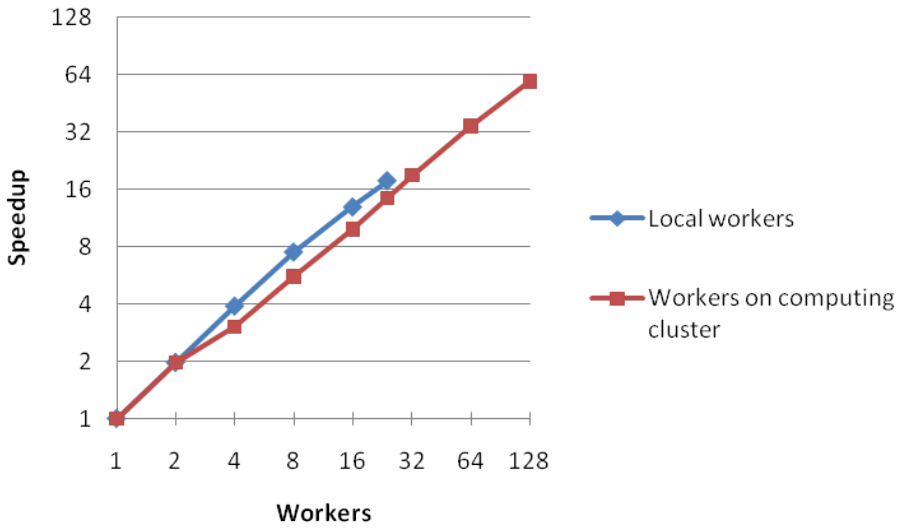
The genome used is hg18 (the human genome) and the bin size is 1mbp.

We see that the speedup achieved is fairly good. Local workers are more efficient than remote workers on the compute cluster.

For some of the tables, the results have been scaled according to the

Workers	Local workers	Workers on compute cluster
1	1	1
2	0.99	0.99
4	0.97	0.77
8	0.94	0.69
16	0.81	0.61
24	0.74	0.6
32		0.59
64		0.54
128		0.46

(c) Efficiency. Data used is from (b).



(d) Plotting the data in (b). Note the use of the logarithmic scale on both axes.

Figure 5.3: Speedup versus number of workers for Usage scenario 2: TFs vs. diseases, both with workers running on the HyperBrowser machine and on the compute cluster. For reference, the single-threaded original implementation uses 1391 seconds on this analysis.

efficiency or speedup of a single worker. For example, for Figure 5.3b, every number in the column for local workers has been divided by 1.95, as that was the speedup achieved for a single worker. This has been done to mask away somewhat strange results so that the most important thing is focused on; how well the system scales.

Chapter 6

Discussion

6.1 Analysis of results

6.1.1 Framework results

The framework example results are mainly intended to identify the maximum throughput Parallel Python and the task queue can sustain. The serial speed of the same program is not all that interesting; however, for reference is takes about 1369 seconds for the 3

We can see that for all jobs with 10^5 tasks, none go under ~ 105 seconds. It seems like the system can handle a maximum throughput of roughly $\frac{10^5}{105} \approx 950$ tasks per second. We can assume that this is a limitation of Parallel Python's internal queue. As jobs connect to the task server through a pipe, it could also be the pipe that is the limitation. This seems unlikely, though; pipes are generally very fast.

This limit of 950 tasks per second could prove a problem for analyses with very many relatively small tasks, such as Monte Carlo runs. Usage scenario 1 (section 5.3.1) for example, has 16 bins (chromosome 17 is 81mbp, the bin size is 5m, and the first 3mbp are excluded from the analysis) of 20000 resamples each. With the default granularity of 100 resamples agglomerated per task, this yields 3200 tasks. If the bin size was reduced to 1mbp there would be five times as many. However, the actual computation time on the worker side easily eclipses the time required for Parallel Python to push the tasks out to the workers. If the amount of tasks do end up becoming a bottleneck, it will most likely be a configuration issue rather than an issue with PP; the task size should be revised rather than blaming PP.

In addition to identifying the maximum throughput PP can sustain, the results show that for purely compute bound tasks, the framework scales linearly up until at least 128 workers. This is to be expected; if a parallel framework cannot achieve a linear speedup when summing independent numbers, something has gone wrong somewhere.

We see from the results that speedup achieved is fairly good. However, there is a fairly strange anomaly - superlinear speedup for a *single* worker.

6.1.2 Superlinear speedup

We see that for both Usage scenario 1: Histone modifications vs. SINE repeats (section 5.3.1) and Usage scenario 2: TFs vs. diseases (section 5.3.2) there are superlinear speedups. This is not all that uncommon; however, it *is* uncommon that a parallel program with a *single* processing element performing work is faster than the serial version. For Usage scenario 1, it only a few percent and as such could be explained by cache effects (this is however fairly unlikely). A more likely explanation is the startup cost issue described in section 6.3.1 — the workers only have to perform the relatively expensive imports the first time they compute a task, not for subsequent ones. As these results are averaged over many runs, the startup cost is masked away.

However, for Usage scenario 2, the speedup with a single local worker is nearly 2. This is not explainable by "normal" explanations for superlinear speed increases. In fact it cannot even be considered a superlinear speed increase as that implies it only happens when code is run in parallel. Skipping imports like described above can at most attribute a modest few percent to a long run like this. To uncover the reason behind this discrepancy, the runs were profiled and compared against each other. The somewhat surprising is that for certain NumPy calls (namely, `reduce` on `numpy.ufunc` objects), the worker runs much faster - for the serial implementation these calls take 10ms on average, while for the parallel implementation it takes 3ms. Finding a plausible explanation for this has not succeeded. One cautious suggestion is that it could be related to the same issue that required a change from using the `subprocess` module to using `multiprocessing` in Parallel Python (presented in section 4.3) — NumPy calls were very slow in processes spawned with `subprocess`. However, this explanation does not seem very plausible, as then the same speedup should be observed for the workers running on the compute cluster. See Section 7.7.2.

6.1.3 Usage scenario 1: Histone modifications vs. SINE repeats

From Usage scenario 1: Histone modifications vs. SINE repeats we can see that the Monte Carlo solution enjoys a nearly linear speedup from adding more workers, at least locally. The workers on the computing cluster do not scale as well as the local workers - this is to be expected due to the increased communication costs incurred when sending tasks to workers on the other end of a network consection. We can also see that the speedup drops significantly past 16 nodes, at least for the computing cluster. This is

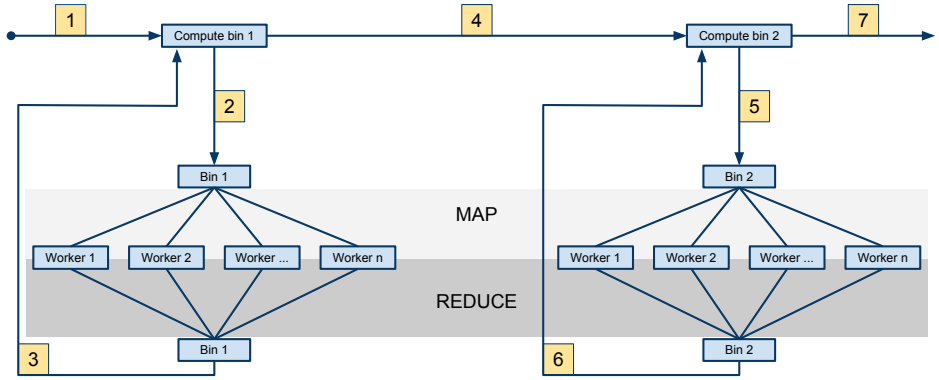


Figure 6.1: An example of the program flow in a small, two-bin Monte Carlo analysis. The numbers represent the flow.

to be expected, due to how the Monte Carlo statistic is implemented.

The Monte Carlo implementation in the Hyperbrowser is designed in a way that necessitates a series of alternating parallel-serial phases, where a very small serial portion is required in between each parallel phase. Because of this the extra communication costs that using the computing cluster entails are exacerbated. This design has fairly obvious flaw. Given that there is more than one bin in the run, performing the parallelization here results in a series of parallel - serial phases, perhaps more easily understood if explained as several map - reduce phases. For each bin, there is a map/reduce step, but these are not done in parallel; each bin is calculated one after another , even if the computation done for each bin *is* done in parallel. See Figure 6.1.

Startup For each map phase, there is some communication delay involved in sending tasks to the workers. During this communication delay, the workers will be idle as they are waiting for tasks.

Finalization For each reduce phase, there will be communication delay as the results are gathered from the workers. In addition, as workers probably do not finish their tasks at the exact same time, some worker time will be spend idling as the computation cannot continue until *all* workers are finished and all results are gathered.

More workers, more delay This is an issue with most parallel algorithms. As more workers are added, the initial delay increases, as more workers will be standing idle for longer during the initial task distribution phase. Let n be the number of workers and t be the number of tasks. As

task 0 is sent to worker 0, worker 1 to n is idle, then as task 1 is being sent to worker 1, worker 2 to n is idle, and so on.

This is aggravated with the MC solution implemented here, as this worker idle time occurs for each initial task distribution phase per bin.

6.1.4 Usage scenario2: TFs vs. diseases

The results from Usage scenario 2: TFs vs. diseases are good. However, considering the fact that the analysis has no extra complexity (like the alternating serial-parallel phases of the Monte Carlo analysis described above), the results are somewhat disappointing. The efficiency drops significantly as the amount of workers is increased.

Some further testing showed that this is due to a required serial phase at the end of the analysis - the global analysis takes a fair amount of time, possibly as the number of bins is high (1Mbp bins over the entire genome yields over 3000 bins) and it has to combine the results from all of these. Testing showed that the global analysis phase for this statistic takes more or less exactly seven seconds for a genome-wide analysis. If this serial portion is subtracted from the results, the results look much better and shows that the reduced efficiency with a significant number of workers is not an issue with the framework, but rather simply Amdahl's law (see section 2.2.3).

Figure 6.2 shows Usage scenario 2: TFs vs. diseases with the seven seconds of necessary serial postprocessing subtracted. The efficiency for 16 and 24 workers on the computing cluster appear somewhat strange, but can can probably be attributed to measurement errors. Regardless, more tests should be run. The issues with getting runtime on the computing cluster for larger jobs prevented further tests.

What is more interesting is the fact that the efficiency is 1 for two workers, before dropping sharply to 0.77 for 4 workers. The same does not happen for local workers. Further tests should be run to uncover why this happens.

6.2 Discussion of design choices

6.2.1 Scheduling of the task queue

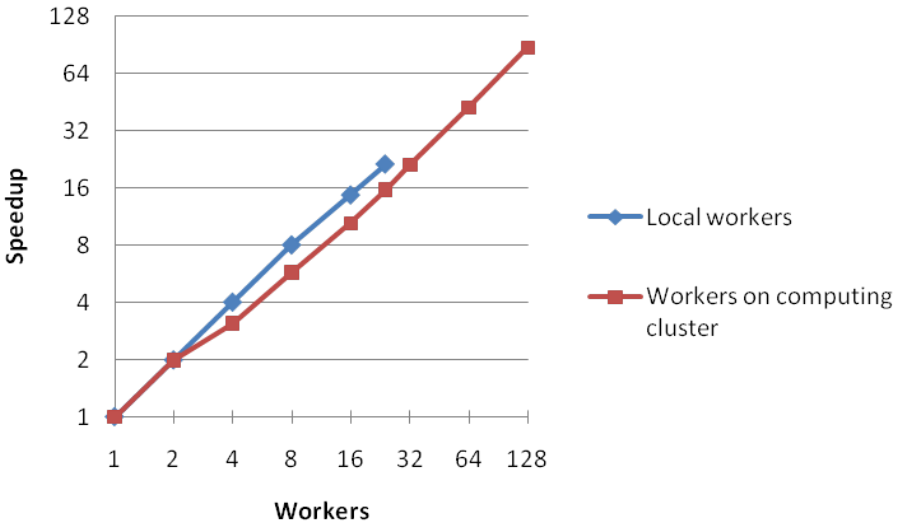
Currently the shared task queue managed by the master task server uses a simple FIFO scheme. This can be problematic if a large job has submitted a large number of tasks and a small job is started later. The large job will occupy all available workers until done as it had submitted its tasks first (even if there is a large amount of workers available, if the job is sufficiently large the small job will stall for a significant amount of time). A lot of analyses are rather small (i.e. they take a few seconds), and a system where small jobs have to wait for hours because a large job is running is not ideal.

Workers	Local workers	Workers on computing cluster
1	1	1
2	1.99	1.99
4	4.01	3.11
8	8.03	5.75
16	14.71	10.44
24	21.4	15.67
32		21.21
64		42.48
128		87.63

(a) Scaled speedup. Both columns have been scaled by the speedup achieved by a single worker.

Workers	Local workers	Workers on computing cluster
1	1	1
2	1	1
4	1.01	0.77
8	1.01	0.72
16	0.92	0.65
24	0.89	0.65
32		0.67
64		0.67
128		0.68

(b) Efficiency. Data used is from (b).



(c) Plotting the data in (b). Note the use of the logarithmic scale on both axes.

Figure 6.2: Results as from Figure 5.3, but modified to take into account the necessary serial postprocessing. In effect this means subtracting seven seconds from the execution time.

A few solutions are presented:

Priority for jobs

Jobs that are somehow known to be small are put in a high priority queue that gets preference over the normal queue. This would solve the issue of small jobs stalling because of a large job occupying the queue. The problem with this is the same that was considered in section 3.1.3): How can we determine the runtime of a job through simply analyzing the job parameters?

Separate queues

Another approach would be to keep separate queues in the master task server for each job, much like it is done in network routers and switches with a queue for each input port. Tasks would be sent to worker processes in a round robin fashion. This suffers from not taking into account that the tasks vary in size (in effect runtime), and it would mean that a job with few tasks with long runtimes would get preference over a job with many tasks with short runtimes.

Timeslicing

An improvement over the previous scheme would be to use something like timeslicing as used in most modern operating system schedulers. Have a separate queue for each job the task server is handling, and share available between them using a system similar to time slicing (exactly how the timeslicing would work is an interesting question in its own part). However, no preempting would have to happen; the time limits can be considered soft. Preemption would in any case be hard to implement as it would involve instructing remote workers to abort execution of a task in the process of being executed. Combining some sort of timeslicing with separate queues for each job would enable somewhat fair sharing of the available resources. This will almost certainly solve the problem in a satisfactory manner. The *multi-level feedback queue* scheduling algorithm closely resembles this and could be the basis for further work; see [19] for a description.

6.2.2 All jobs share the same computing power

With the design choices outlined, all concurrently executing HyperBrowser jobs will share all avilable computing power. However, I would argue that this is the most beneficial solution for many reasons.

The main reason is efficient utilization of compute cluster allocations. When all jobs share all the available computing power, allocations on the compute cluster can be fully exploited. If each job was to have its own, separate allocation on the compute cluster, some of the time allocated would

probably be wasted, as all jobs must finish before the time they have allocated is over. By sharing the computing power, this is minimized.

Second, the only thing that should matter is total throughput of the system. As sharing the computing resources available should hopefully improve this compred to each job having its own allocations, this is a good enough reason in itself.

6.2.3 Interactive jobs

A key point of the HyperBrowser is that the jobs are interactive, and thus it is hard to determine their runtime. This has been attempted solved through the design by exploiting the fact that the HyperBrowser jobs are embarassingly parallel, and hence can be split into completely independent parts. This is then exploited through the use of relatively short-lived compute cluster allocations. This again exploits the fact that Parallel Python supports workers joining and leaving at arbitrary points in time without being adversely affected; tasks are simply rerouted if the worker assigned to it disappears. This could not have been done for a problem where each task had to

This design works fairly well. Not knowing the runtime of the job beforehand is no problem as the jobs are not affected by "losing" tasks during computation, due to no communication being disrupted.

6.2.4 Partition

The chosen partition as described in section 3.3.1 has many benefits, chief among them that it was relatively easy to implement and generates tasks with trivial communication requirements. However, it does have one major drawback: it is only effective if the analysis used has at least as many bins as workers, or if the analysis uses a Monte Carlo model. Currently there are very few analyses for which this is not the case. In the future, though, the HyperBrowser team is planning on implementing machine learning methods in order to develop models for the relationship between tracks; this would be computationally intensive for single bins. The partition here would not work very well for such an analysis. Another partition would have to be implemented — perhaps one where each statistic is considered a task could be used, as discussed briefly in Section 3.3.1

6.2.5 Building a queueing system on top of another queueing system

The system presented has a certain strangeness to it — it is essentially implementing a shared queueing system on top of another shared queueing system. This is true to a certain degree. However, if one thinks of the HyperBrowser jobs not as a number of discrete jobs, but rather as a whole

— a single entity with computational requirements that vary widely over time — it makes more sense.

6.3 Discussion of implementation details

6.3.1 Overhead

Parallel Python Parallel Python, as noted in section 2.4.3, performs a lot of internal work on a per-task basis (type and argument checking, scheduling as well as caching of the actual method that is to be performed (see section 2.4.3. This naturally adds some overhead to the parallel implementation, as is to be expected. However, the amount of overhead Parallel Python adds could almost certainly be improved. For example, the method caching functionality of Parallel Python is, for the Hyperbrowser, entirely unnecessary. There is no need of being able to send any given method (actual source code) to the workers as a full Hyperbrowser install is always available at the worker. Currently, the tasks sent to the workers are designed in such a way that the function sent is as small as possible; this minimizes the initial overhead (as less source code has to be transferred).

Workers on the computing cluster When using workers running on the computing cluster, some extra overhead is added. This is to be expected. A task going to a worker on the computing cluster has to go through three communication channels: First through a pipe (technically a `multiprocessing.Consection` object, but this uses an operating pipe internally) from the job to the shared task queue, then through a network consection (socket) from the task queue to the task server handling the worker, then through another pipe from the task server to the worker. Both pipes and sockets are relatively fast, but some extra time is unavoidable.

Initial startup cost

When sending tasks to the computing cluster (and to a certain degree to local workers), there are some initial setup costs involved (not related to the actual network communication cost or workers idling during the initial communication phase as described above). In effect, this makes getting the results from the first task take much longer than the rest. This is mostly related to Python. The most time-consuming initial overhead is simply importing the necessary libraries. For example, importing NumPy alone takes from 1.5-2 seconds. When this is done only once per job, the amount of time taken is negligible. When it is done for each worker, however, it can become significant; this initial startup cost in effect becomes an unavoidable serial part of the program, something which is avoided as much as possible for parallel programs due to Amdahl's law. Do note, though, that this only

happens when a new worker connects to the task server and thus only for the first task any worker computes — for any subsequent tasks, the startup cost has already been taken care of. This makes the cost appear high when looking at separate, relatively short jobs, but for real-world usage scenarios these startup costs will be negligible. An additional five seconds of computation time for the first task any worker computes does not matter much for overall throughput when a worker can be reasonably expected for live for at least an hour.

6.3.2 Rpy

An issue that has not yet been mentioned is that of R. R[1] is an open-source statistical software environment for statistical modelling and the HyperBrowser uses it for some of its statistics. To interface with R, rpy[2] is used. Rpy is a simple Python interface to R.

However, rpy has a bug that causes R to crash (with the not too helpful error message `"C stack usage is too close to the limit"`)if it is run from a program that uses threads. Threads are used by Parallel Python to communicate with workers (so that the program does not block while waiting on network or pipe Input/Output (I/O)). This means that, currently, the statistics that use Rpy quite simply cannot be run.

Preliminary testing shows that rpy2 (a rewrite and redesign of rpy) does not exhibit this problem. As a migration to rpy2 is in the works by the HyperBrowser team, effort has not been expended on working around or fixing this issue.

[1] http://www.r-project.org/
[2] http://rpy.sourceforge.net/

Chapter 7

Future work

7.1 Automatic allocation of titan jobs

The current implementation of the computing cluster job submission functionality (described in section 4.4) is not very good. Either compute cluster jobs are manually submitted with a command line tool, or the automatic bakground service solution can be used. However, the background service is currently only a proof-of-concept — simply inspecting the queue every n minutes and submitting a job asking for more task server on the compute cluster if the queue found to not be empty. Clearly a better load metric than this must be devised if this is to be put into real day-to-day use. Simple, better metrics would be the load average for all currently available workers for the last n minutes. or even "are there more than x tasks in the queue"). No explicit cancelling of remote task servers that are running while the queue is empty would really be necessary: if the job allocations are kept somewhat "short" (as in, they have a wallclock limit of an hour or so), they will simply die off on their own fairly.

7.1.1 Suggestion for an improved automatic allocation scheme

Have each job report anticipated time left to the host task server, based on the time to retrieve the last results. The job must calculate this itself as the task server has no notion of how many tasks a job consists of — all it does is distribute tasks and retrieve results as fast as possible. If all tasks had been submitted once and only once per job the task server could be made to calculate this, but for example for MC jobs, tasks are submitted in several batches, making the task server calculate a projected finish time for the entire job a difficult prospect. In addition, the amount of tasks a job has is currently always known a priori, but may change in the future if the number of resamplings used in a MC job is dynamically determined during the job execution to arrive at a given p-value. This is a planned future improvement that the HyperBrowser team is in the process of implementing.

If the cumulative anticipated time left exceeds some threshold, request more computing power. How much computing power that is to be requested can be static (like the current implementation which always requests one node), or it can be determined based on how much the threshold has been exceeded (how much each worker can do in a given time unit is fairly easy to calculate). Note that this algorithm would not work very well with the current implementation, as all jobs submits their tasks to a shared, FIFO queue. Therefore only the job with its tasks first in the queue would have any data to base its predictions on. The suggested improvement (described in section 6.2.1) that makes each job have its own queue managed by the shared task queue supervisor would make this viable, as each job would get fair slices of computing time.

7.1.2 Low priority queue

Related to this is the low priority compute cluster queue. As shown, the remote workers running on the compute cluster do not have to be long-lived. If they go down while computing a task, the task will simply be allocated to another available worker. Because of this, running a large number of task servers on the Titan low priority queue (fittingly named "lowpri") should be a viable way of increasing the available computing power. Jobs running in the lowpri queue can (and will) often be killed as the nodes they run on are commandeered by other jobs with higher priority, but as shown this is not a problem. This will also prevent idle workers running on the compute cluster from taking up valuable computing resources on doing nothing — if the nodes are required for more important work, they will be killed by the queueing system.

7.2 Improved handling of random number generation

As mentioned in section 4.5, a way of supplying specified seeds on a per-task basis has been implemented. However, a way of generating these seeds in a good way has not. All the RNGs used in the libraries that utilize random numbers (R, NumPy and the Python runtime) use the same algorithm for generating their random numbers, namely the Mersenne Twister [22]. As for each task, the RNG is seeded with a number that is (possibly) quite similar to another task's seed, their streams are not guaranteed to be muturally independent. How important this is for statistical tests is not really my field of expertise, nor is number theory. Nonetheless, implementing the scheme shown in [21] for mutually independent RNG streams might be an interesting candidate for future work.

7.3 Inspection of the Titan queue and better overview of the queue as a whole

All jobs submitted to SLURM file under a specific account. These accounts are used for bookkeeping and have limits associated with them, such as processor hours per year and maximum number of processors in use at the same time. Especially the maximum number of cores available to the project simultanously is of interest to us. Submitting more job allocations to the compute cluster if the capacity has already been reached is pointless.

At the moment the ability to inspect the job queue on Titan is minimal at best. This is mainly because interfacing with SLURM is not all that easy; it is done via shell commands. Writing a parser for these was not deemed important enough to warrant the time it would have taken. Regardless, the information that is obtainable with these shell commands would be very useful. For example, checking how many cores that is actually currently available for the project which the HyperBrowser submits its jobs under. This information could then be used to improve the automated batch script submission functionality, see section 4.4.1. Currently jobs are automatically started without paying any attention at all to how many cores are currently available to the project, which is of course not ideal.

Another useful addition would be the ability to inspect the queue to try and anticipate when a job would receive an allocation. This has very limited support in SLURM (and even for SLURM, anticipating it is hard as a job may be moved in the queue due to backfilling (see section 2.3.1).

7.4 Checkpointing

There is currently no support for checkpointing of jobs running on the HyperBrowser. Do note that neither did the original, serial implementation either. However, it would be a useful feature in case of unforeseen occurences like power outages or the host computer unexpectedly going down. Because of the pickling functionality implemented so that results can be transferred from the workers back to the host computer, implementing checkpointing should be a relatively straight forward affair.

7.4.1 Suggested checkpointing solution

As explained in section 4.1.2, results are returned from the workers in a picklable form. This allows for a simple checkpointing solution: have the job store these results on the disk every n minutes, along with basic statistics such as how many bins that have been computed. As the results are already picklable, no special handling is required. If the host computer goes down unexpectedly, a partial "snapshot" of the job will be stored on disk. When the

job is continued after the host computer starts up again, this snapshot can be used to continue the job from the last checkpoint. This, of course, requires that Galaxy supports the concept of continuing a job after an unexpected shutdown — this has not been investigated.

7.5 Handling crashing jobs with tasks in queue?

A problem that has not been handled is what happens to "orphaned" tasks left in the task queue: What would happen if a job submitted a number of tasks to the queue, then crashed before the results are ready? The tasks would be stuck in limbo as no request would be made to retrieve their results. A solution would be to do a pass over all tasks whenever a job disconnects (times out) and remove the ones that belong to said job.

7.6 Disk caching of results

The HyperBrowser supports disk caching of results, so that for subsequent analyses with the same parameters, results are loaded from disk rather than running the analysis anew for no real reason. This has been disabled during the development of the parallel implementation. Before the system can be put into production, a way of checking for results cached on the disk should be implemented in a way that prunes away tasks that would have been sent to workers for which disk cached results already exist.

7.7 Validity and criticism

7.7.1 Compute cluster results

The results from the usage scenarios where the workers were ran on the computer cluster cannot be guaranteed to be correct. As receiving these allocations could take a very long time (between 12–48 hours for the 64- and 128-worker runs), many of the results were averaged using the same allocation. Thus they ran after one another within a short timespan. As the compute cluster is under variable amounts of load, this could affect the results. For example, perhaps the cluster wide I/O load is higher than average during this short timespan and thus makes the reading of files from the distributed file system slower.

7.7.2 NumPy

NumPy exhibits some strange behaviour, see Sections 4.3 and 6.1.2. Especially the speedup which occurs running a single local worker is strange, and suggests an error somewhere — either in the parallel implementation,

in the original serial implementation, or possibly in NumPy itself. Regardless, it seems safe to say that there is some kind of issue with NumPy and multiprocessing.

Chapter 8

Conclusion

8.1 Summary

Chapter 2 provides necessary background to understand the thesis: The HyperBrowser system, theory on parallel programs, practical information about compute clusters and selected topics from the Python programming language were presented.

Chapter 3 described how the parallel design theory known as PCAM was applied to the HyperBrowser in order to yield a parallel design.

Chapter 4 presented a framework for parallel computation of disjoint tasks, both locally and on nodes on a compute cluster. How the parallel design presented in Chapter 3 was applied to the HyperBrowser was then explained.

Chapter 5 listed the results from some example runs, demonstrating the performance both of the framework on its own as well as when used in the HyperBrowser.

Chapter 6 discusses the results presented in Chapter 5, along with a discussion around the design and implementation choices taken in this thesis.

Chapter 7 presented a number of suggested future improvements to the system.

8.2 Contribution

The defining characteristic of the presented framework is the ability to distribute independent tasks automatically amongst workers spread over many

machines, both locally and on a compute cluster. Through the implementation on the HyperBrowser, it is here shown that this approach can relatively easily be implemented for an existing system, yielding efficient, transparent load balancing across both the local computer system and external compute clusters.

8.3 Findings

Three research questions were raised in the introduction and are discussed below.

8.3.1 Is retroactively parallelizing a large Python system viable?

"Can a large-scale Python program designed without parallelism in mind effectively be parallelized in order to exploit today's multi-processor architectures? If so, can it be done without massive changes to the existing code base?"

For the system presented in this thesis, translating the serial design into a parallel design was not too difficult. The system presented here offers good speedup and scales well. However, do note that the design would not have worked nearly as well had it not been embarassingly parallel, i.e. that the tasks can be easily split into independent tasks that do not require inter-task communication.

The solution presented in this thesis does not modify the existing code in any significant way at all, as much effort has been expended in keeping the parallel code separate from the serial code. In fact, only at three locations in the original implementation has "entry points" to the parallel code been inserted.

8.3.2 Is using a compute cluster a viable way of speeding up execution times

"Is offloading work to a high-performance compute cluster a viable way of speeding up execution times and increasing the available computing power? If so, can it be done in a way transparent to the user?"

Offloading tasks to a computer cluster is demonstrated to work very well. The idea of using the compute cluster to run short-lived remote workers that come and go seems to be a good one. Like in the previous research question, it would not work very well if the problems are not embarassingly parallel. However, many problems can be decomposed into independent subproblems, and for those this approach yields excellent results.

In addition, the implemented system uses the compute cluster without the users knowing about it.

8.3.3 Can an interactive system efficiently exploit a compute cluster?

"Compute clusters are designed for maximum throughput, rather than quick response. Can an interactive system where small jobs run alongside larger jobs still yield short execution times for the small jobs when using a computer cluster?"

This research question was formulated somewhat later than the rest. Therefore we can only give a tentative "yes". Currently, small jobs in a light load scenario will return their results quickly. However, under heavy load, if a small job starts after a large job, it will have to wait until the large job is finished. A solution to this is presented in Section 6.2.1 that will solve this — the groundwork is there, all that is required is the implementation of the suggested algorithm.

Glossary

bin datasets are divided into bins, usually on a per-chromosome basis. The size of the bins is defined by the user. These bins are used in the local analysis step to be able to avoid drawing incorrect conclusions from the global analysis.

broadcasting a networking term for sending a message to a specific network address so that it reaches every computer on the same network segment.

host computer the computer which runs the HyperBrowser.

job refers to a problem submitted to the HyperBrowser, typically a statistical analysis of some sort.

NumPy Python package for scientific computing which, among other things, supports a powerful array programming model.

pipe a First-In-First-Out communication cahnnel used for Inter-Process Communication.

task a subproblem of a larger problem encapsulated in such a way that it can execute independently from and in parallel with other tasks.

task server a component of Parallel Python, which manages workers and distributes tasks amongst these for computation.

track in this thesis, short for "genomic annotation track", a collection of objects of a specific genomic feature, such as genes, with base-pair specific locations for the entire genome.

worker a program that runs in its own process which receives tasks from its local task server, computes them, and returns the result.

Acronyms

bp base pair.

FIFO First-In-First-Out.

GIL Global Interpreter Lock.

GPU Graphics Processor Unit.

I/O Input/Output.

IPC Inter-Process Communication.

MC Monte Carlo.

PCAM Partitioning, Communication, Agglomeration, Mapping.

PP Parallel Python.

RNG Random Number Generator.

SLURM Simple Linux Utility for Resource Management.

SMP Symmetric MultiProcessing.

SPMD Single Process, Multiple Data.

Appendix A

Framework

A.1 An example of how to use the framework

Let us use the summation problem as an example of how the framework works. The problem is to sum all the numbers from 0 to n. This is naturally parallelizable; simply have each task sum a subrange of the sequence, then sum these in a final "reduce" phase. It can also be parallelized with a recursion based scheme for even better performance, but we will not do that here.

To demonstrate, the problem has been somewhat simplified. We let each task simply sum a range of numbers. In fact, each task sums the *same* range of numbers. To do this, we implement two wrappers: one that handles the splitting of the problem and the merging of the results, and one that describes the work that is to be done by the workers.

The example is written to be run from the command line, simply use `python Sum.py n m`, where n is the number of tasks that should be computed and m determines how many numbers each task should sum; from 0 to 10^k.

Listing A.1: Using the presented framework to solve the summation problem.

```
Line 1  from JobHandler import JobHandler
    -   import sys
    -   import time
    -
    5   class SumJobWrapper(object):
    -       def __init__(self, n, taskSize):
    -           self.n = n
    -           self.taskSize = taskSize
    -
   10       def __iter__(self):
    -           for i in xrange(0, self.n):
    -               yield 0, self.taskSize
    -
```

```
-     def handleResults(self, results):
15        return results
-
- class SumTaskWrapper(object):
-     def handleTask(self, task):
-         start, end = task
20        return sum(xrange(start, end))
-
- if __name__=='__main__':
-     import quick.application.parallel.Sum
-     n = int(sys.argv[1])
25    taskSize = int(sys.argv[2])
-     startTime = time.time()
-     wrapper = quick.application.parallel.Sum.\
-         SumJobWrapper(n, 10**taskSize)
-     jobHandler = quick.application.parallel.Sum.\
30        JobHandler("dummyId", useSharedTaskQueue=True)
-     result = jobHandler.run(wrapper)
-     print "summed %d numbers, took %f seconds." %\
-         (int(n), time.time() - startTime)
```

A note: the import in `__main__` may seem unnecessary, but is required due to the way Python handles module paths. When distributing the tasks, the fully qualified class name (i.e. an unambigous reference to the class that does not vary regardless of scope) of the task wrapper is sent to the worker so that it may correctly import it. However, if the explicit import of the wrappers is not performed like in the example, both wrappers would be bound to `__main__`, and extracting the fully qualified class name would result in `__main__.SumTaskWrapper`, which would obviously not be importable at the worker.

Appendix B

Initial implementation

> Plan to throw one away, you will anyhow.
>
> Fred Brooks

The overall Hyperbrowser architecture is very complex, and was not fully understood when work was begun on this thesis. During the initial analysis it seemed that all that had to be done to achieve a significant speedup would be to parallelize at the local analysis stage in the job execution. Therefore, the initial implementation simply spawned a predetermined number of processes (using the `multiprocessing` module) that would run the `doLocalAnalysis` method in `StatRunner`. Each bin would be enumerated and the processes would compute the results for $\frac{b}{n}$ bins, where b is the number of bins and n the number of processes. The results from each bin were stored in a list; only the results that were explicitly returned from the `getSingleResult` were used. The results were returned in list form (via a `multiprocessing` queue) to the main process which combined them into one list and then continued with the global analysis and result report generation. This approach was flawed for a number of reasons:

- Only returning the results from the top level statistic is not enough. It works for certain simple statistics, but as explained in Section 2.1 often the global analysis performed at the end of the run requires the results from not only the top level statistic, but statistics further down in the hierarchy.
- Each worker process would work on an equal, predetermined number of bins. This is not a very good strategy, as certain bins can take much longer to compute than others. Often one process would take significantly longer than the others to finish its workload.
- Only gives a significant speedup if there are many bins; if an analysis of a small part of the genome is performed as is common (like a Monte

Carlo analysis on a part of a chromosome) is being performed the runtime will not improve significantly.

At this point it was thought the implementation worked fine, quite simply due to bad testing practices. The tests that were being run at the time were all simple statistics that gave the correct results with only the results from the top level statistic passed into the global analysis. A queue-based solution was implemented to make work distribution better, like in a standard master/slave (or task farming) setup. A concurrent queue (from multiprocessing) was used. The bins were enumerated, their index put on the queue, and the worker processes would fetch an index from the queue before perfoming work on the corresponding bin. This approach worked relatively fine. Still, many issues were present. Testing this implementation revealed that simply returning the top level results in list form was simply an entirely wrong solution. It became clear that a entirely new way of returning results was needed when it was realized that much of the results were actually being stored transparently in statistic objects in the MagicStatFactory cache.

The current implementation of how the memoized results stored in `MagicStatFactory` are pickled and returned to the master process was then developed.

After a working results collection solution had been implemented, a very basic compute cluster solution was developed, using MPI for communication. It worked in a rather crude manner; a master process and n worker processes were allocated on the compute cluster (using manually written batch scripts, no helper functionality had yet been devised). Once they started up, the master process read a file from disk that contained instructions on what to do, i.e. a description of the `StatJob` with track names, statistic name and so on. Tasks were then handed out to worker processes with a standard manager/worker setup: the master keeps a list of tasks that need to be computed. The only difference from the standard manager/worker was that the workers would not return the results directly to the master process as they finished them. Instead they were kept in the worker's memory (in the `MagicStatFactory` cache), and when the computation was finished it would serialize the entire `MagicStatFactory` to a file on the disk. The results files from the workers would then be read by the job at the HyperBrowser server and used as the job's `MagicStatFactory` cache, as if it had been computed locally. Most of the current implementation that relates to submitting jobs to the computing cluster comes from this implementation.

While this worked, it had numerous drawbacks:

- Returning results in file form is slow and not very elegant
- An entire process (the master) that is mostly idle while occupying a compute cluster processor
- Computation would not start until the job allocation had been filled as there was no support for combining local and remote workers

In addition it still had the issues of not taking into account Monte Carlo jobs and being too tied to existing code base.

At this point it was realized that doing all of this was fairly pointless when there existed open-source packages that could handle this part of the system in a much better way. As a wise man once said, whenever possible, steal code.

Appendix C

Source code

The source code can be downloaded at

`http://jonathal.at.ifi.uio.no/master/`

The code is also part of the HyperBrowser project, and will be included in an upcoming release. This can be downloaded at

`http://hyperbrowser.uio.no`

Bibliography

[1] Hyperbrowser team. What is the Genomic HyperBrowser? `https://sites.google.com/site/hyperbrowserhelp/?tool_id=hb_help` [Online; accessed 17-April-2011].

[2] OGF DRMAA Working Group. Distributed Resource Management Application API. `http://www.drmaa.org/` [Online; accessed 25-February-2011].

[3] Poznan Supercomputing and Networking Center. PSNC DRMAA for SLURM. `http://apps.man.poznan.pl/trac/slurm-drmaa` [Online; accessed 25-February-2011].

[4] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[5] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. Above the clouds: A berkeley view of cloud computing. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28*, 2009.

[6] D. Blankenberg, G. V. Kuster, N. Coraor, G. Ananda, R. Lazarus, M. Mangan, A. Nekrutenko, and J. Taylor. *Galaxy: A Web-Based Genome Analysis Tool for Experimentalists.* John Wiley & Sons, Inc., 2001.

[7] S. H. Bokhari. On the mapping problem. *IEEE Trans. Comput.*, 30:207–214, March 1981.

[8] P. D. Coddington. Random number generators for parallel computers. *The NHSE Review*, 2, 1996.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[10] M. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901 – 1909, dec. 1966.

[11] I. Foster. *Designing and Building Parallel Programs.* Addison-Wesley, 1995.

[12] M. Fowler. *UML distilled: a brief guide to the standard object modeling language.* Addison-Wesley Professional, 3rd edition, 2004.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: elements of reusable object-oriented software.* Addison-Wesley, 1995.

[14] J. Goecks, A. Nekrutenko, J. Taylor, and T. G. Team. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome Biology*, 11(8):R86, 2010.

[15] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing,.* Pearson, 2 edition, 2003.

[16] C. P. U. Group. Inside the python gil. From a workshop on Python concurrency for the Chicago Python User Group, 06 2009.

[17] J. Ioannidis, D. Allison, C. Ball, I. Coulibaly, X. Cui, A. Culhane, M. Falchi, C. Furlanello, L. Game, G. Jurman, et al. Repeatability of published microarray gene expression analyses. *Nature genetics*, 41(2):149–155, 2008.

[18] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72. ACM, 2007.

[19] L. Kleinrock and R. Muntz. Processor sharing queueing models of mixed scheduling disciplines for time shared system. *Journal of the ACM (JACM)*, 19(3):464–482, 1972.

[20] M. Mascagni and A. Srinivasan. SPRNG: A scalable library for pseudorandom number generation. *ACM Transactions on Mathematical Software-TOMS*, 26(3):436, 2000.

[21] M. Matsumoto and T. Nishimura. Dynamic creation of pseudorandom number generators. *Monte Carlo and Quasi-Monte Carlo Methods*, pages 56–69, 1998.

[22] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

[23] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[24] G. K. Sandve, S. Gundersen, H. Rydbeck, I. Glad, L. Holden, M. Holden, K. Liestol, T. Clancy, E. Ferkingstad, M. Johansen, V. Nygaard, E. Tostesen, A. Frigessi, and E. Hovig. The genomic hyperbrowser: inferential genomics at the sequence level. *Genome Biology*, 11(12):R121, 2010.

[25] G. K. Sandve, S. Gundersen, H. Rydbeck, I. Glad, L. Holden, M. Holden, K. Liestøl, T. Clancy, F. Drabløs, E. Ferkingstad, M. Johansen, V. Nygaard, E. Tøstesen, A. Frigessi, , and E. Hovig. The differential disease regulome. Resubmitted, *BMC Genomics*.